Informatique et Calcul Scientifique

Cours 8 : Algorithmique

16.04.2025

La fois passée, on a vu..

Le module `NumPy` qui :

- permet d'effectuer des opérations mathématiques et statistiques vectorisées sur un nouveau type d'objets, les `ndarray`

Le module `Matplotlib` qui :

- permet la représentation graphique de données ou de fonctions
- le chargement ou la création d'images

Ces deux librairies s'utilisent souvent ensemble et sont extrêmement répandues dans le monde scientifique.

But de la leçon

Aujourd'hui on...

Aujourd'hui, on parlera science de l'algorithmique :

- Comment évaluer les performances d'un programme/algorithme ?
- La notation de Landau

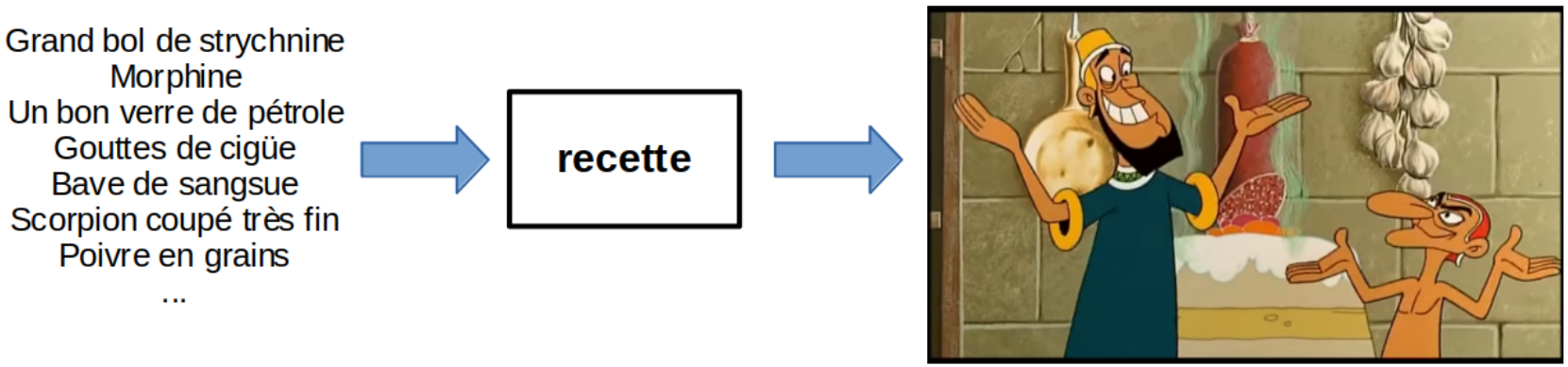
Il s'agit de la mise en pratique de nos connaissances en programmation !

Qu'est-ce qu'un algorithme ?

Un algorithme est une **procédure** pour résoudre un problème.

- Il prend en **entrée** (**input**) une **instance** de ce problème
- et produit la **sortie** (**output**) correspondant à cette instance.

On peut assimiler un algorithme à une recette de cuisine [1].



1. comme celle du pudding à l'arsenic (remix)

Qu'est-ce qu'un algorithme ?

Par exemple, on peut écrire un algorithme qui calcule le minimum d'une liste de nombres.

- **Entrée** : une liste de nombre réels
- **Sortie** : le minimum des nombres de la liste
- **Algorithme** (en français, ou pseudo-code) :
  1. Définir `my_min` comme le premier élément de la liste
  2. Parcourir chaque élément de la liste. S'il est plus petit que `my_min`, mettre `my_min` = cet élément
  3. Retourner `my_min`. Celui-ci correspond donc au plus petit élément de la liste.

Qu'est-ce qu'un algorithme ?

Une instance possible de ce problème est la liste `[10, -3, 7, 2]` :

`[10, -3, 7, 2]` $\Longrightarrow$ calculate_min $\Longrightarrow$ `-3`

Une autre instance serait la liste `[8, 33, 5, -20, 0]` :

`[8, 33, 5, -20, 0]` $\Longrightarrow$ calculate_min $\Longrightarrow$ `-20`

Ne sont pas des instances valides de ce problème :

- `['a', 2, 'c']`
- `[[0, 1], [1, 2]]`

Algorithme vs. implémentation

Remarquons qu'à ce stade nous n'avons pas présenté d'implémentation d'un algorithme.

- Un **algorithme** est une procédure générale pour résoudre un problème.
- Un **programme** est une implémentation spécifique d'un algorithme dans un langage de programmation donné, par exemple Python, et sur un système donné.
- A un même algorithme correspondent donc plusieurs implémentations [2].
- Les algorithmes vus dans ce cours seront donnés sous forme de programmes Python.

2. Voir http://www.rosettacode.org pour des implémentations de divers algorithmes en plus de 800 langages de programmation.

Problème vs. algorithme

A un problème peuvent correspondre plusieurs algorithmes qui le résolvent

- *Comment caractériser "le meilleur" ?*

Pour certains problèmes on ne sait pas s'il existe un algorithme qui les résoud [3] ; pour certains problèmes on sait qu'il n'en existe aucun [4].

Pour certains problèmes, on ne connaît encore aucun algorithme efficace qui les résoud.

- *Que veut dire "efficace" ?*

3. https://en.wikipedia.org/wiki/Collatz_conjecture
4. https://en.wikipedia.org/wiki/Halting_problem

Critères d'évaluation d'un algorithme

Il y a plusieurs manières de déterminer l'efficacité d'un algorithme.

- **Correctitude**. L'algorithme doit calculer ce qu'il est censé calculer ! Pour cela, on utilise un invariant de boucle (non étudié dans le cadre de ce cours).
- **Performance**. La performance d'un algorithme s'évalue selon différents critères :
  - Le temps de parcours
  - Espace requis en mémoire
  - Nombre d'appels à la mémoire de disque
  - ...

Dans la suite, nous nous concentrerons sur l'étude du temps de parcours.

Analyse du temps de parcours

On s'intéresse au temps de parcours d'un algorithme **en fonction de la taille de l'entrée**

- Si l'entrée est un objet possédant n éléments *de taille fixe*, on dira que l'entrée est de taille n

On suppose en général que les opérations suivantes prennent **un temps constant** :

- Opérations arithmétiques : addition, soustraction, multiplication, division, reste entier,...
- Manipulation de données : créer une variable, affecter une valeur à une variable, lire et comparer les valeurs de deux variable,...
- Opérations de contrôle : instructions `if` ,...
- Appel d'une fonction
- Accéder à un élément d'une liste `L[i]` étant donné l'index `i`
- Invoquer la fonction `len()` , et les méthodes `append()` et `pop()` (sans arguments !) sur une liste.

Exemple : rechercher le maximum d'une liste

Reprenons l'exemple précédent et comptons le nombre d'opérations effectuées.

```
def max_liste(L):                    c0

    n = len(L)                       c1
    max_L = L[0]                     c2

    for i in range(1, n):            c4
        if L[i] > max_L:             c5   } n-1 itérations
            max_L = L[i]             c6

    return max_L                     c3


L = [0, 3, 10, -7, 5]
max_liste(L)
```

L'appel à la fonction `max_liste()` prend un temps $T(n)$

$$T(n) = c_0 + c_1 + c_2 + c_3 + (c_4 + c_5 + c_6)(n-1) = c + c'n.$$

La fonction `max_liste()` a un temps de parcours **linéaire** en n.

Mesurer le temps de parcours

La fonction `time()` du module `time` de Python retourne le temps au moment de l'appel en secondes, calculé depuis une date de référence qui dépend du système (souvent le 1er janvier 1970).

- On l'utilisera pour mesurer le temps pris par un appel à la fonction `max_liste`.

La fonction `randrange(start, stop, step)` du module `random` retourne un nombre aléatoire entre `start` et `stop` (on utilisera `randrange(N)` pour un grand entier `N`).

- On l'utilisera pour générer une liste `L` de grande taille qu'on passera en argument à `max_liste`.

On peut finalement utiliser la librairie `Matplotlib` pour représenter le temps pris par l'algorithme en fonction de la taille de l'instance.

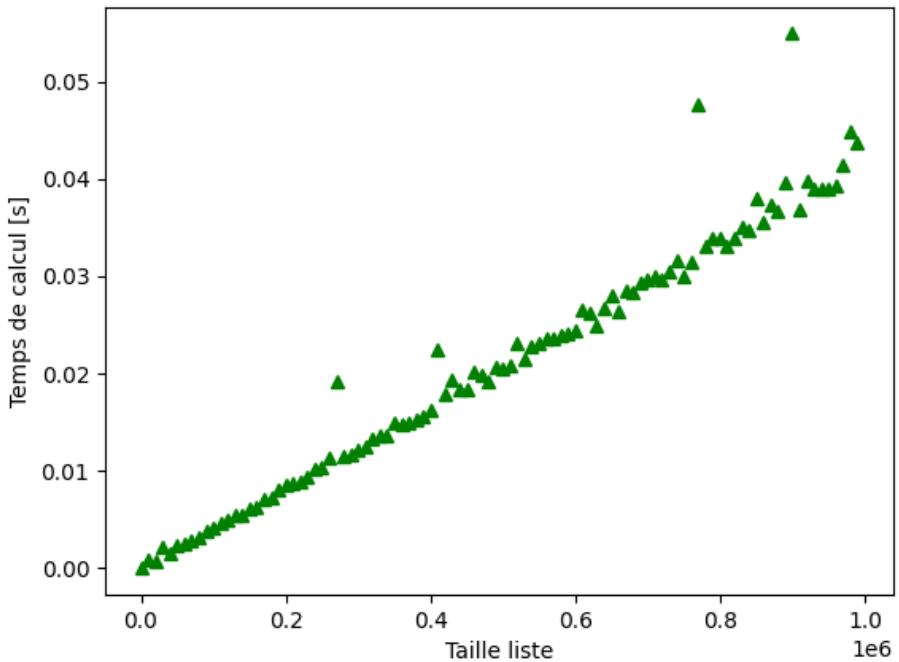
Mesurer le temps de parcours

```python
import numpy as np
from time import time

def max_liste(L):
    n = len(L)
    max_L = L[0]
    for i in range(1, n):
        if L[i] > max_L:
            max_L = L[i]
    return max_L

N = 1000
a = np.random.rand(N)

t0 = time()
m = max_liste(a)
t1 = time()
print(f"L'appel a pris {t1-t0} s")
```

Mesurer le temps de parcours

Exercice : testez le temps de parcours de l'algorithme `max_liste` en changeant la taille de la liste `L` .

- Ci-dessous : les temps de parcours empiriques observés pour l'appel de `max_liste()` en fonction de la taille de la liste fournie en entrée.



Exemple : somme maximale

Etant donnée une liste de n nombres ($n \geq 2$), donner un algorithme qui calcule la plus grande somme de deux éléments d'indices distincts de la liste.

Par exemple,

- **Entrée :** `L = [945, 815, 1132, 731, 981, 673]`
- **Sortie :** `1132 + 981 = 2113`

Exemple : somme maximale

La première méthode serait de parcourir toutes les paires d'éléments `(i,j)` possibles.

```
def max_somme(L):
    '''
    Entree: liste L de nombres de taille n >= 2
    Sortie: Somme maximale de deux elements de L
    '''
    n = len(L)
    max_s = L[0] + L[1]

    for i in range(n):
        for j in range(i+1, n):
            if L[i] + L[j] > max_s:
                max_s = L[i] + L[j]

    return max_s
```

On appelle ce type d'algorithme qui essaie toutes les combinaisons possibles un **algorithme de force brute**.

Temps de parcours

Question : Combien de paires `(i,j)` distinctes existe-t-il ?

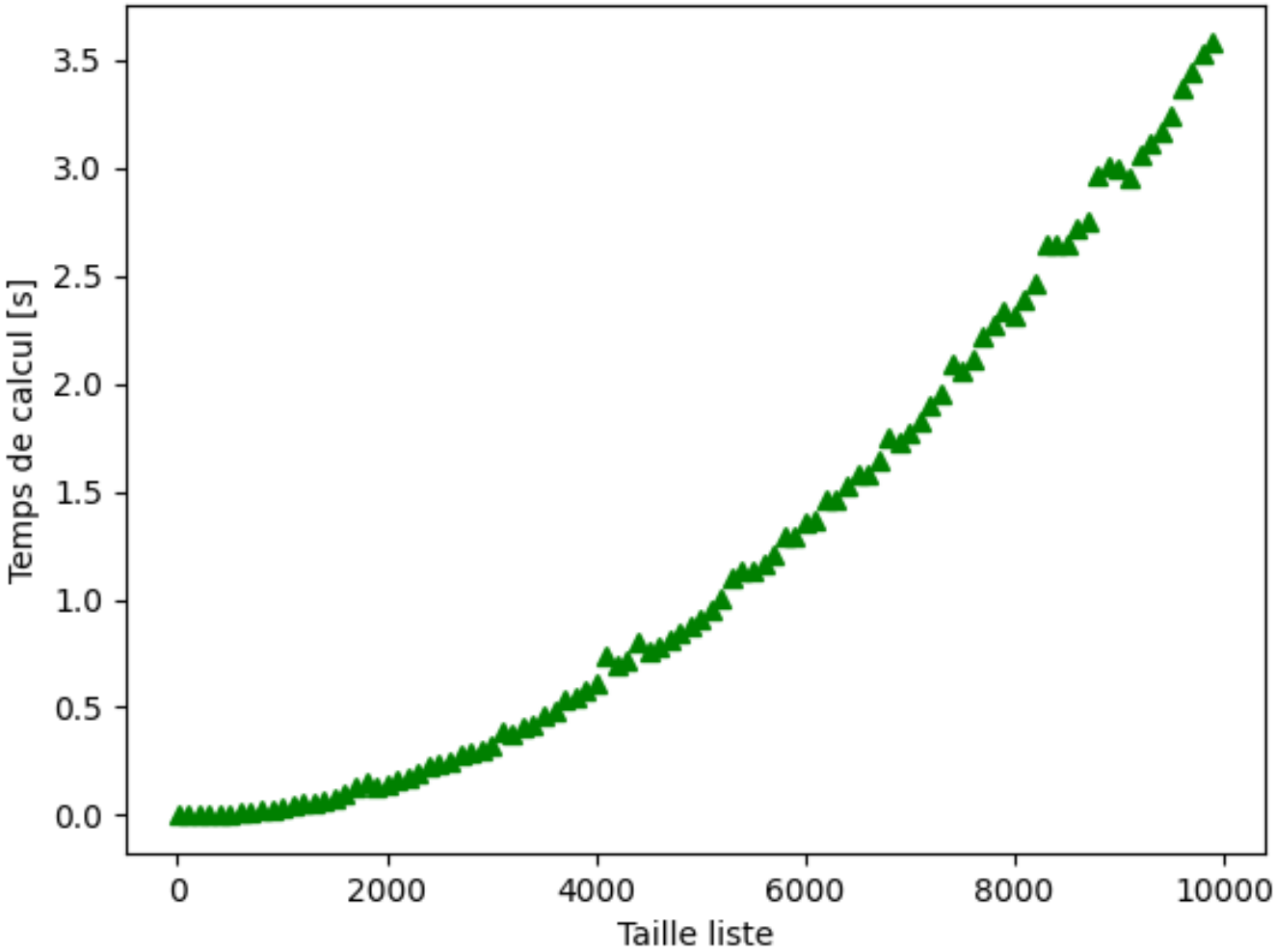
```
for i in range(n):
    for j in range(i+1, n):
        if L[i] + L[j] > max_s:
            max_s = L[i] + L[j]
```

- La boucle extérieure est exécutée n fois (`i = 0, ..., n-1`).
- A la `i`ème itération de la boucle extérieure, la boucle intérieure est exécutée $n - i - 1$ fois :
  - **`i = 0`** : `j` parcourt `range(1, n)` : $n - 1$ itérations
  - **`i = 1`** : `j` parcourt `range(2, n)` : $n - 2$ itérations
  - ...
  - **`i = n-1`** : `j` parcourt `range(n, n)` : 0 itérations
- Temps de parcours des boucles imbriquées :

$$c \cdot [(n-1) + (n-2) + \cdots + 1 + 0] = c \cdot \frac{n(n-1)}{2}.$$

- Temps de parcours de l'algorithme : $T'(n) = c_2 n^2 + c_1 n + c_0$.

Temps de parcours



Somme maximale : un autre algorithme

Existe-t-il un autre algorithme plus performant pour résoudre le même problème ?

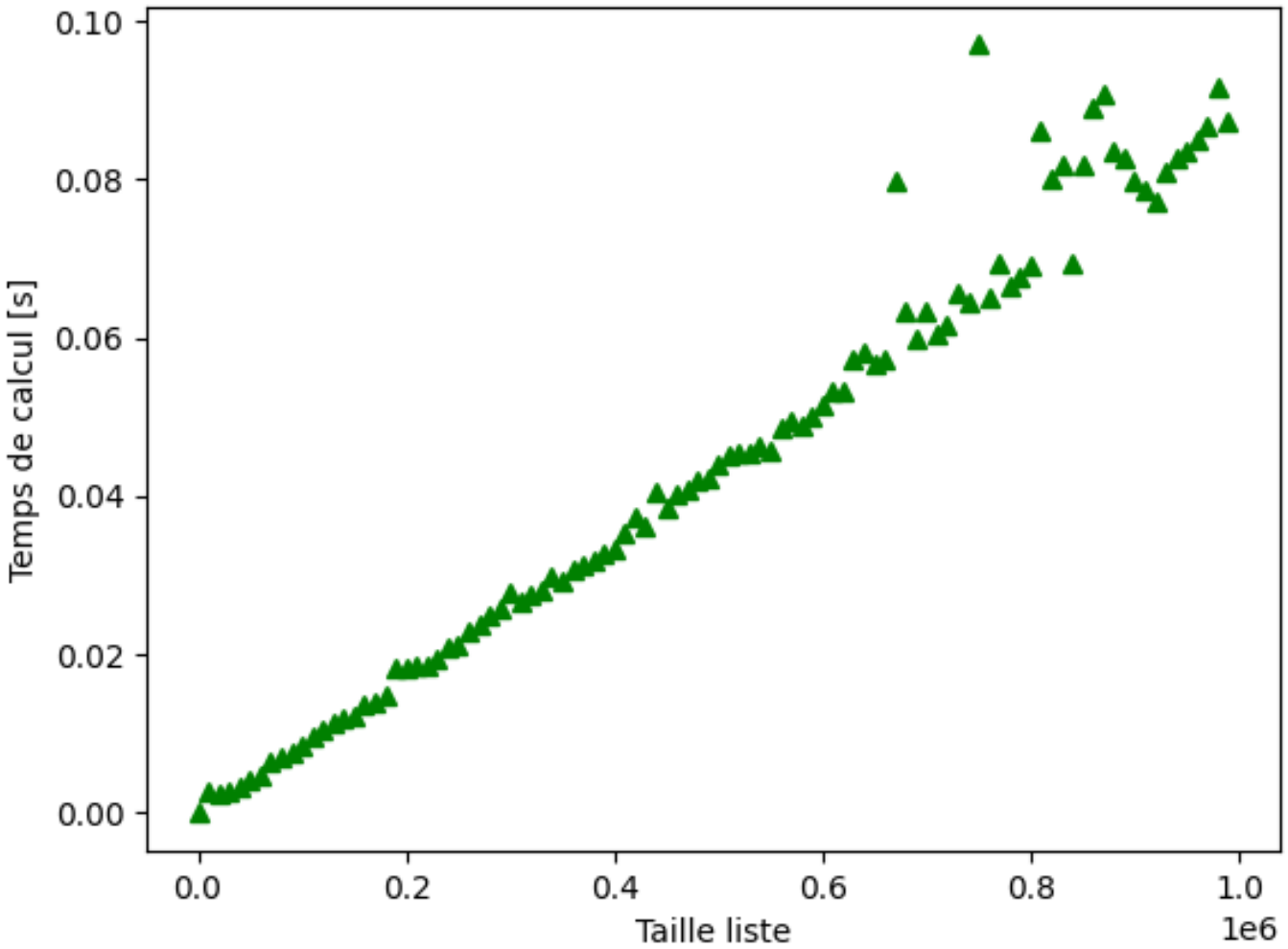
- On peut chercher les deux plus grands éléments de la liste `L`. Leur somme donne le résultat souhaité.

```
def max_somme_lineaire(L):
    '''
    Entree: liste L de nombres de taille n >= 2
    Sortie: Somme maximale de deux elements de L
    '''
    n = len(L)
    max1 = max_liste(L)
    L.remove(max1)
    max2 = max_liste(L)
    return max1 + max2
```

Temps de parcours : $T''(n) = c_3 n + c_4$
si on admet [5] que `L.remove()` a temps de parcours linéaire en n.

5. Sinon, modifiez `max_liste` pour qu'elle rende les deux plus grands éléments de la liste d'un coup, et vérifiez que son temps de parcours est toujours linéaire en n.

Temps de parcours



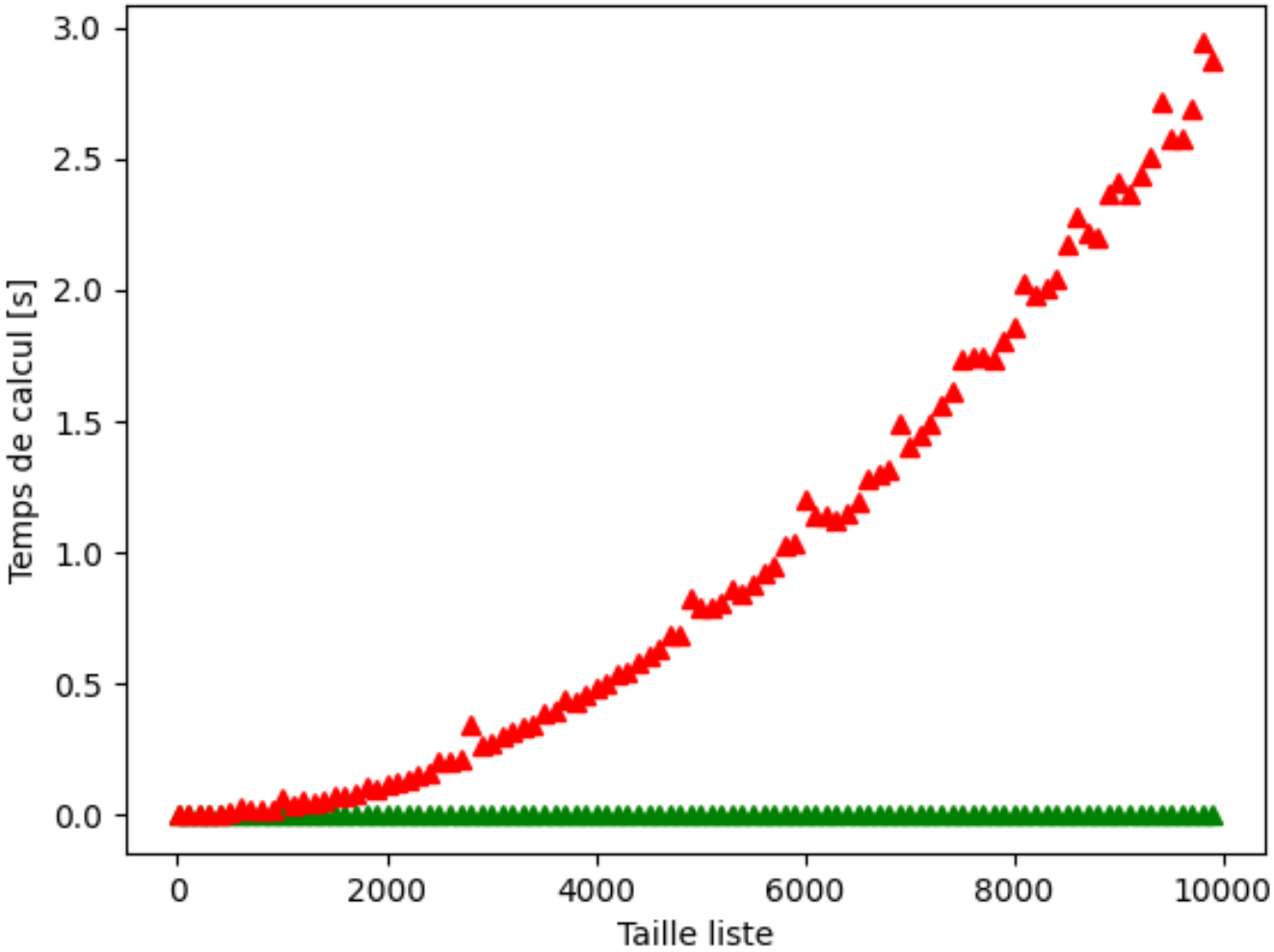
Somme maximale : le meilleur algorithme ?

Quel est l'algorithme le plus performant, `max_somme` ou `max_somme_lineaire` ?

- Exécutez le code ci-dessous pour différentes tailles de liste `L`.

```
import numpy as np
from time import time

N = 1000
a = np.random.rand(N)

t0 = time()
max_somme_lineaire(L)
t1 = time()
print("l'appel a pris", t1 - t0, "secondes")

t0 = time()
max_somme(L)
t1 = time()
print("l'appel a pris", t1 - t0, "secondes")
```

Temps de parcours



Notation $\mathcal{O}(\cdot)$

Le calcul du temps de parcours avec le module `time` est problématique : le temps de parcours varie d'un langage de programmation à l'autre, d'une machine à l'autre, d'un moment à l'autre sur la même machine...

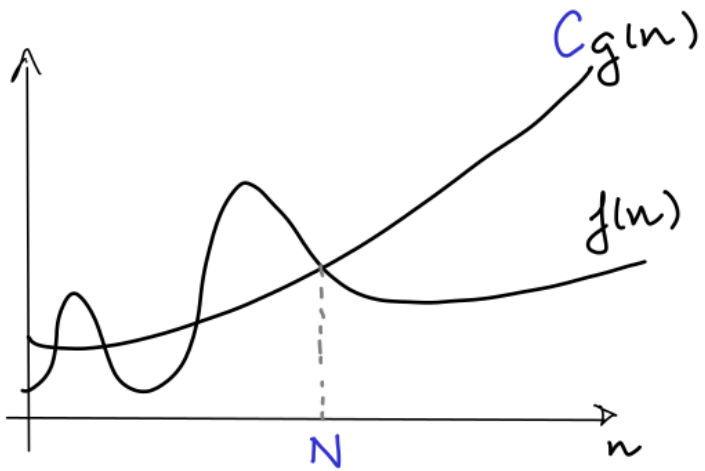
- Pour évaluer la performance d'un algorithme indépendamment des détails d'implémentation, on utilisera la **notation de Landau** $\mathcal{O}(\cdot)$: c'est un outil mathématique qui permet de caractériser **la vitesse asymptotique de croissance d'une fonction**.
- On s'intéressera au **comportement asymptotique** d'un algorithme, *i.e.*, à son temps de parcours en fonction de la taille n de l'entrée **lorsque** n **tend vers l'infini**.

Notation $\mathcal{O}(\cdot)$

Définition (Grand O)

Soient $n \in \mathbb{N}$, et f, g des fonctions positives de n. On dira que "f est $\mathcal{O}(g)$" ou "$f = \mathcal{O}(g)$" s'il existe des réels $C > 0$, $N > 0$ tels que

$$\forall n > N \quad f(n) \leq C \cdot g(n).$$



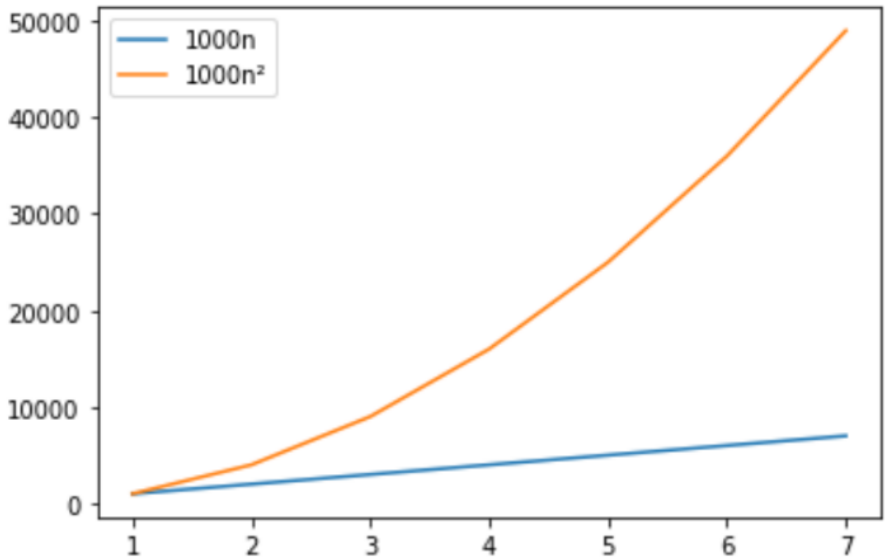
Reformulation : f est $\mathcal{O}(g)$ si $\limsup_{n\to\infty} \dfrac{f(n)}{g(n)} \leq C$.

Exemples

$$f = \mathcal{O}(g) \iff \exists C, N > 0 \text{ t.q. } \forall n > N \quad f(n) \leq C \cdot g(n).$$

- $1000n = \mathcal{O}(n^2)$:

$$1000n \leq \underbrace{1000}_{C} n^2 \text{ pour tout } n \geq \underbrace{1}_{N}.$$

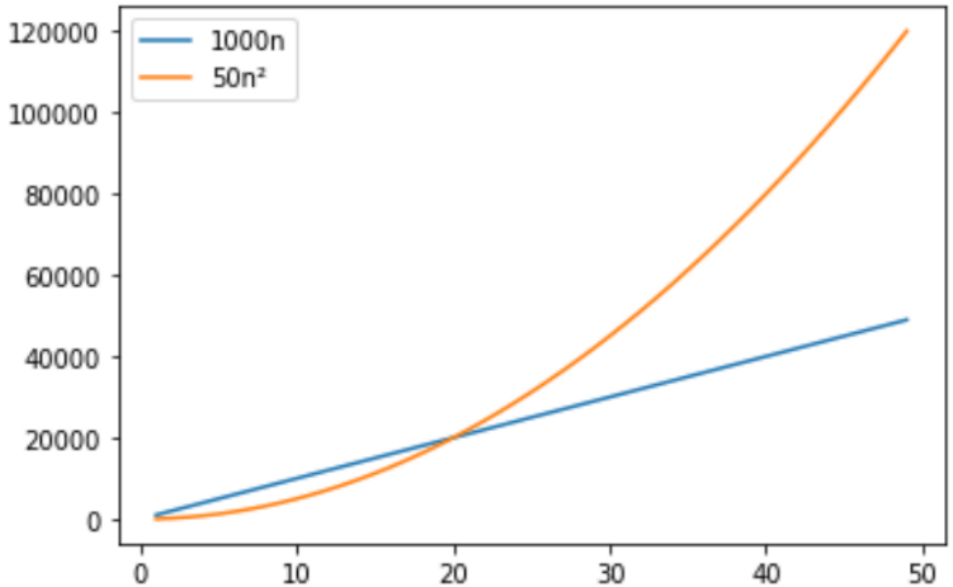


Exemples

$$f = \mathcal{O}(g) \iff \exists C, N > 0 \text{ t.q. } \forall n > N \quad f(n) \leq C \cdot g(n).$$

- $1000n = \mathcal{O}(n^2)$:

$$1000n \leq \underbrace{50}_{C} n^2 \text{ pour tout } n \geq \underbrace{20}_{N}.$$

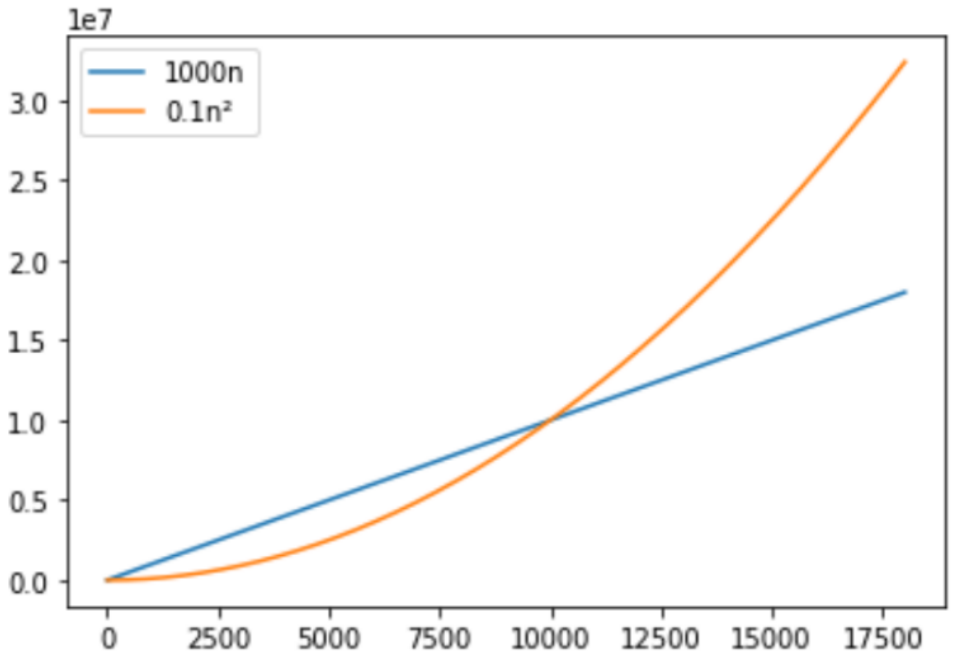


Exemples

$$f = \mathcal{O}(g) \iff \exists C, N > 0 \text{ t.q. } \forall n > N \quad f(n) \leq C \cdot g(n).$$

- $1000n = \mathcal{O}(n^2)$:

$$1000n \leq \underbrace{0.1}_{C} n^2 \text{ pour tout } n \geq \underbrace{10000}_{N}.$$



Puissances rationnelles

$$f = \mathcal{O}(g) \iff \exists C, N > 0 \text{ t.q. } \forall n > N \quad f(n) \leq C \cdot g(n).$$

En général, pour des puissances rationnelles $p \leq q$,

$$n^p = \mathcal{O}(n^q)$$

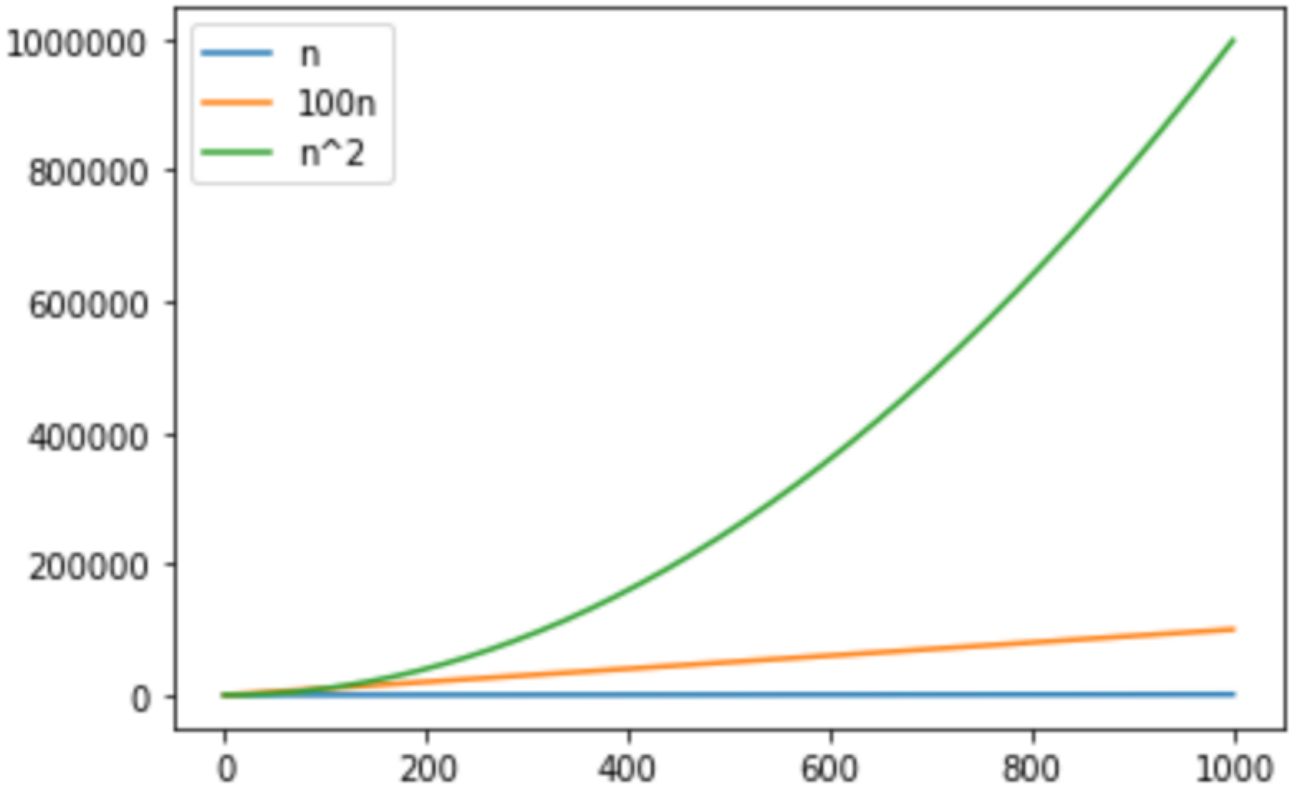
- $n = \mathcal{O}(n)$, $n = \mathcal{O}(n^2)$
- $n^2 = \mathcal{O}(n^3)$
- $\sqrt{n} = \mathcal{O}(n)$
- ...

Si $p < q$, alors $n^p = \mathcal{O}(n^q)$ mais n^q n'est pas $\mathcal{O}(n^p)$.

La relation $n^p = \mathcal{O}(cn^p)$ est valide pour toute constante $c > 0$.

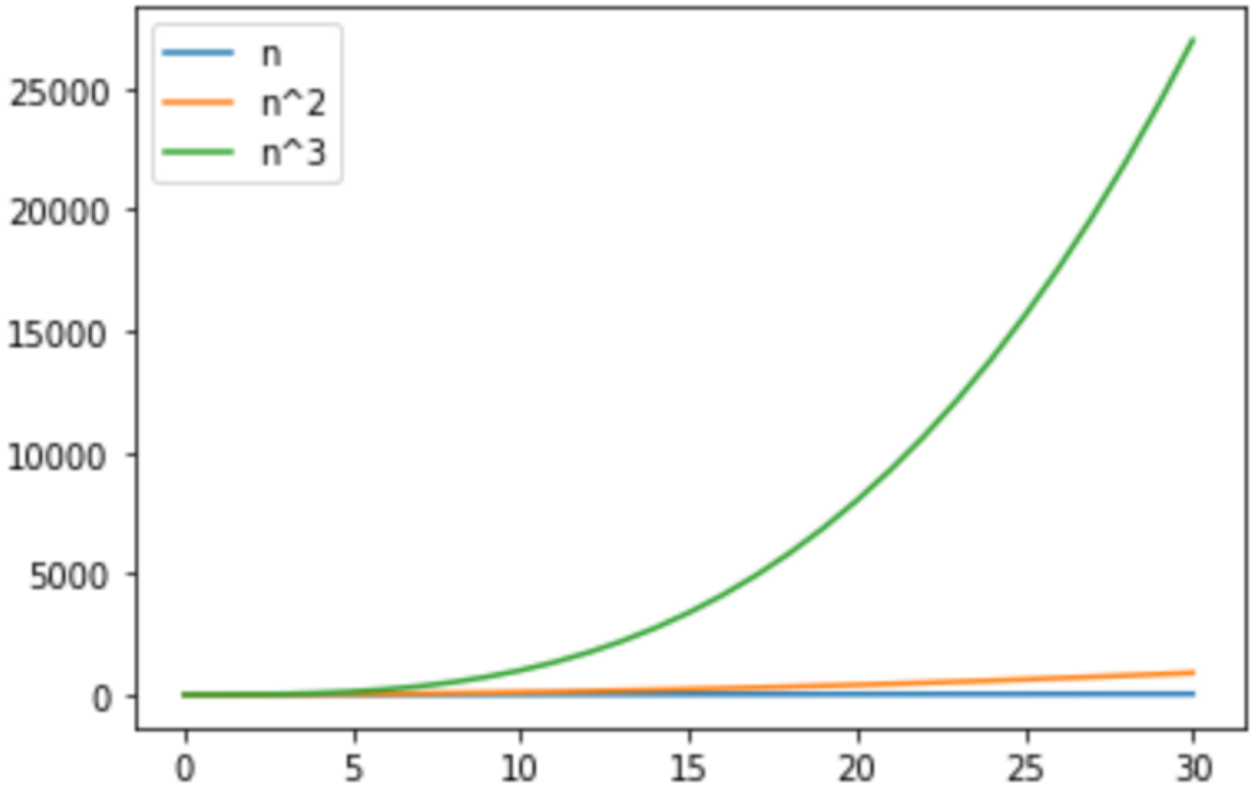
Exemples

$$n = \mathcal{O}(100n) = \mathcal{O}(n^2)$$



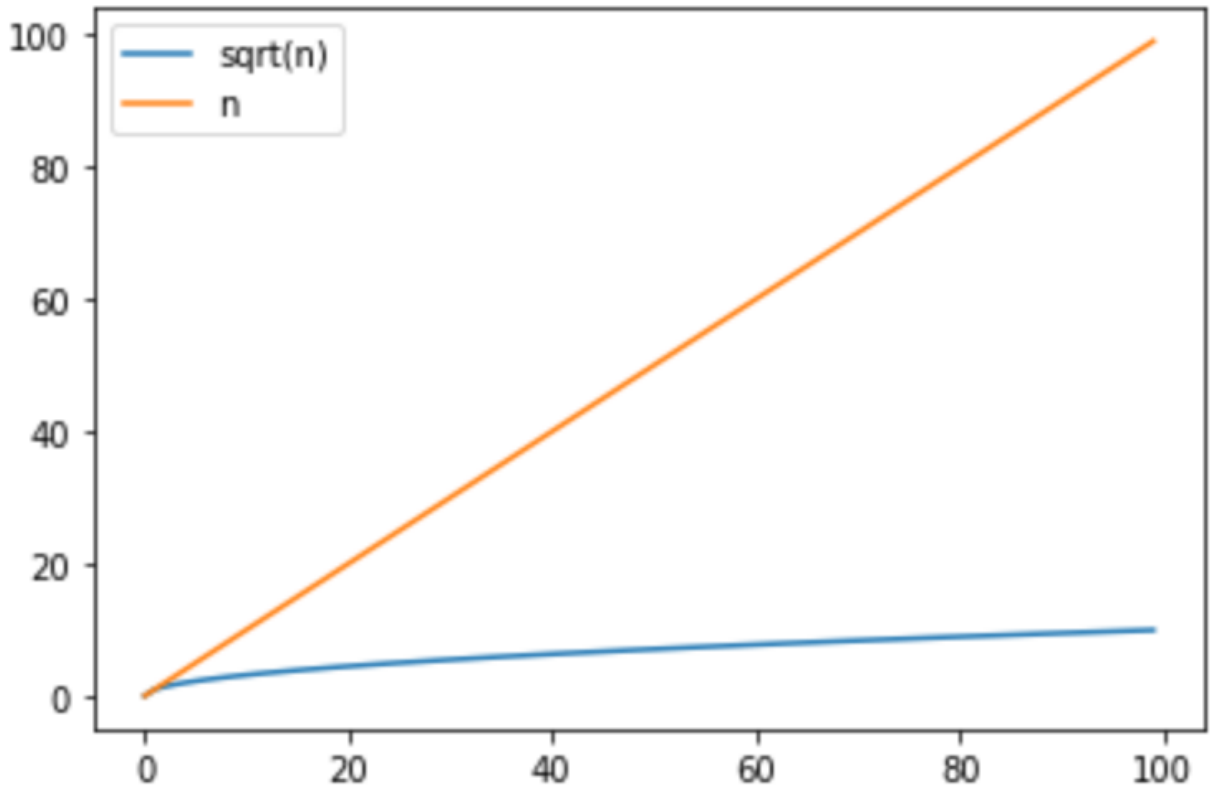
Exemples

$$n = \mathcal{O}(n^2) = \mathcal{O}(n^3)$$



Exemples

$$\sqrt{n} = \mathcal{O}(n)$$



Polynômes

$$f = \mathcal{O}(g) \iff \exists C, N > 0 \text{ t.q. } \forall n > N \quad f(n) \leq C \cdot g(n).$$

Pour $p \leq q$ rationnels ou entiers, f, g polynômes de degré p, q respectivement,

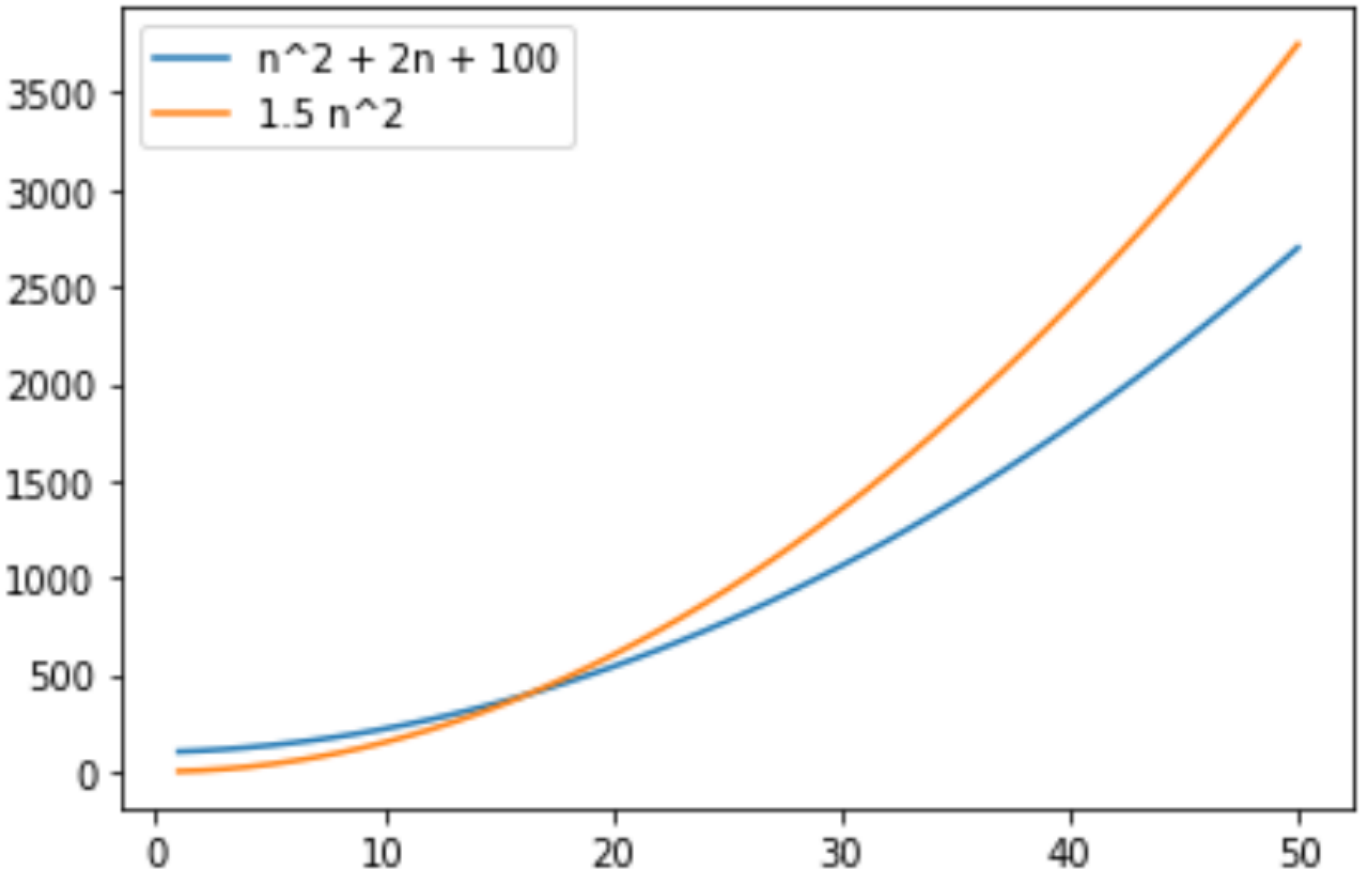
$$f(n) = \mathcal{O}(g(n))$$

- $2n + 100 = \mathcal{O}(n^2)$
- $2n + 100 = \mathcal{O}(n)$
- $n + 10^6 = \mathcal{O}(n)$
- $n^2 + 1000n + 10^6 = \mathcal{O}(n^3)$
- $n^2 + 1000n + 10^6 = \mathcal{O}(n^2)$
- $n + \sqrt{n} = \mathcal{O}(n)$
- $n\sqrt{n} + 1000n = \mathcal{O}(n^{1.6})$
- $\ldots$

Si $p < q$, alors $f = \mathcal{O}(g)$ mais g n'est pas $\mathcal{O}(f)$.

Exemples

$$n^2 + 2n + 100 = \mathcal{O}(n^2)$$



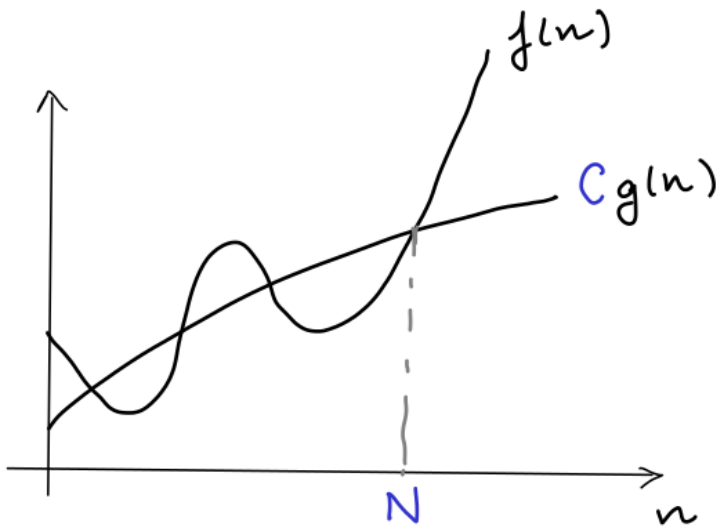
Notation $\Omega(\cdot)$

Définition (Grand Omega)

Soit $n \in \mathbb{N}$, et f, g des fonctions positives de n.
Si $g = \mathcal{O}(f)$, on dira que $f = \Omega(g)$.

Une définition équivalente :

$$f = \Omega(g) \iff \exists C, N > 0 \text{ t.q. } \forall n > N \quad f(n) \geq C \cdot g(n).$$



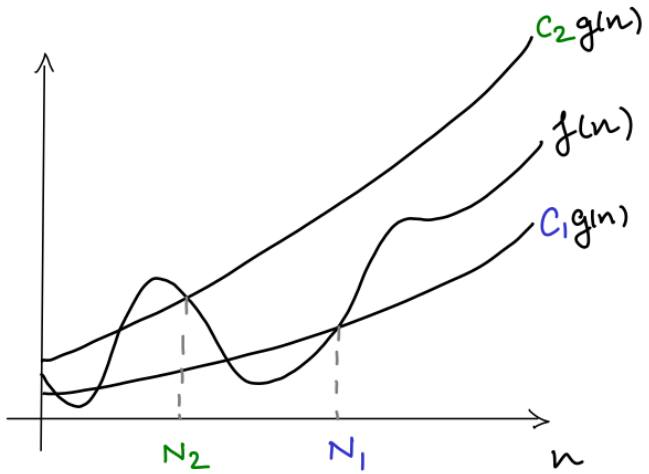
Notation $\Theta(\cdot)$

Définition (Grand Theta)

Soit $n \in \mathbb{N}$, et f, g des fonctions positives de n. Si $f = \mathcal{O}(g)$ et $g = \mathcal{O}(f)$, on dira que $f = \Theta(g)$ (et $g = \Theta(f)$).

Une définition équivalente :

$$f = \Theta(g) \iff \exists C_1, C_2, N > 0 \text{ t.q. } \forall n > N \quad C_1 \cdot g(n) \leq f(n) \leq C_2 \cdot g(n).$$



Exemples

- Pour tous polynômes f et g de même degré,

$$f(n) = \Theta(g(n))$$

- Pour toutes sommes f et g de puissances rationnelles avec le même terme dominant,

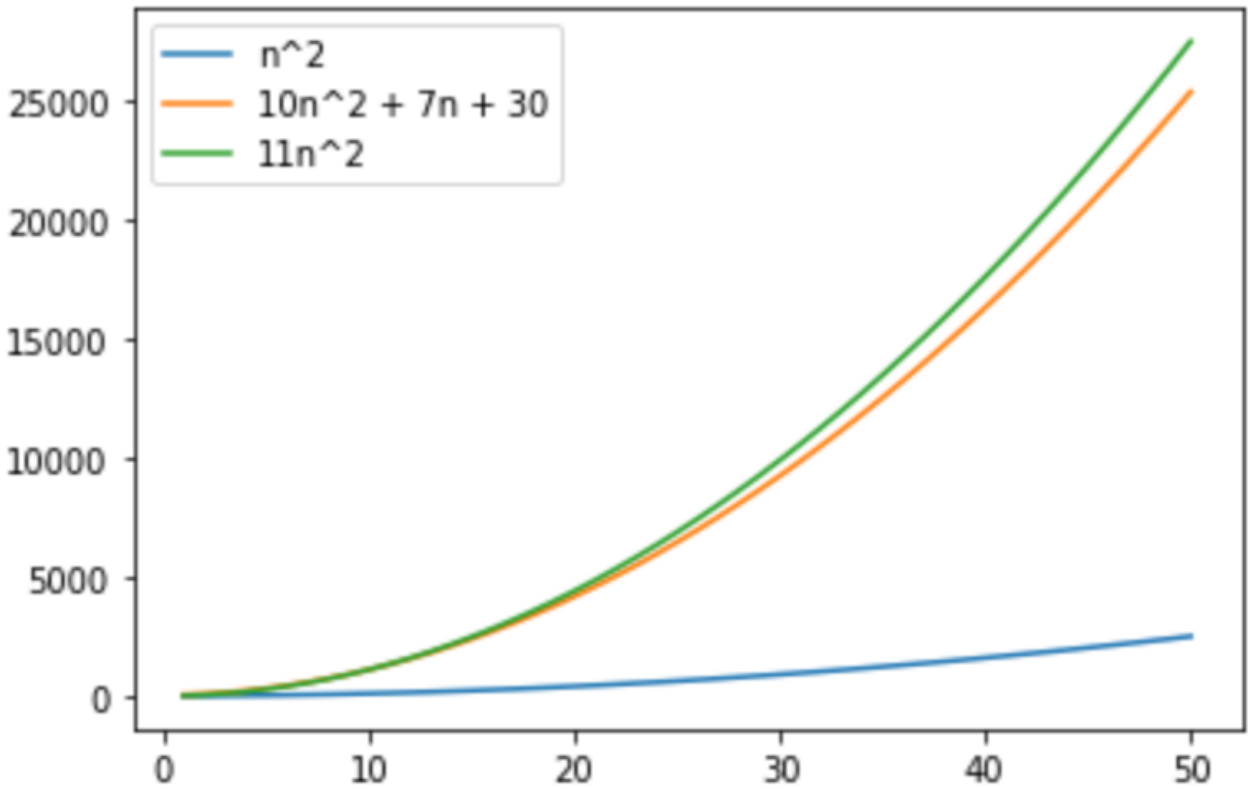
$$f(n) = \Theta(g(n))$$

  - $10n^2 + 7n + 30 = \Theta(n^2)$
  - $1000n^2 + 42 = \Theta(n^2)$
  - $100n\sqrt{n} = \Theta(n\sqrt{n} + n) = \Theta(n\sqrt{n})$
  - $\ldots$

La notation $\Theta(\cdot)$ cache les constantes et les termes d'ordre inférieur, et donne le comportement asymptotique d'une fonction de n (lorsque n tend vers l'infini).

Exemples

$n^2 = \Theta(10n^2 + 7n + 30) = \Theta(10n^2) = \Theta(11n^2) = \Theta(n^2)$ etc...



Temps de parcours d'algorithmes en notation asymptotique

On aimerait exprimer le temps de parcours d'un algorithme comme une fonction $T(n)$ de la taille n de l'entrée, puis utiliser la notation de Landau pour estimer la **vitesse de croissance** de $T(n)$.

- Problème : pour la taille n de l'entrée fixée, le temps de parcours d'un algorithme peut également dépendre de l'instance du problème qui lui est fournie en entrée !
  - Par exemple, problème du tri (voir semaine prochaine)
  - Pour les trois algorithmes de ce cours (`max_liste` , `max_somme` , `max_somme_lineaire`), le temps de parcours est indépendant de l'instance du problème fournie en entrée.

En général, pour un algorithme donné, on définit $T(n)$ comme le temps de parcours de cet algorithme sur une instance de taille n **au pire des cas**.

Borner le temps de parcours d'un algorithme **au pire des cas** offre une garantie sur le temps de parcours.

Temps de parcours d'algorithmes en notation asymptotique

Soit $T(n)$ le temps de parcours d'un algorithme pour une entrée de taille n au pire des cas.

- Si on donne une fonction $f_1(n)$ telle que $T(n) = \mathcal{O}(f_1(n))$, f_1 est une **borne supérieure** sur le temps de parcours de l'algorithme.
- Si on donne une fonction $f_2(n)$ telle que $T(n) = \Omega(f_2(n))$, f_2 est une **borne inférieure** sur le temps de parcours de l'algorithme.
- Si on donne une fonction $f(n)$ telle que $T(n) = \Theta(f(n))$, f est à la fois une borne supérieure et une borne inférieure sur le temps de parcours de l'algorithme : elle décrit le comportement asymptotique du temps de parcours.

Comportement asymptotique de `max_liste`

Pour une entrée de taille n, `max_liste` a temps de parcours (au pire des cas) $T(n) = c + c'n$ donc max_liste a un temps de parcours qui est $\mathcal{O}(n)$ (linéaire en la taille de l'entrée).

Question : Peut-on mieux faire (asymptotiquement) ?

- Non ! Tout algorithme qui recherche le maximum d'une liste de nombres doit au moins parcourir toute la liste, et donc une borne inférieure triviale sur le temps de parcours d'un tel algorithme est $T'(n) = \Omega(n)$.

Un algorithme de recherche du maximum avec temps de parcours $\Theta(n)$ est donc asymptotiquement optimal.

Comparaison des algorithmes étudiés dans ce cours

Pour une entrée de taille n :

- `max_somme` a temps de parcours (au pire des cas) $T'(n) = c_2 n^2 + c_1 n + c_0 = \mathcal{O}(n^2)$:
  c'est un temps de parcours **quadratique** en n.
- `max_somme_lineaire` a temps de parcours (au pire des cas) $T''(n) = c_3 n + c_4 = \mathcal{O}(n)$:
  c'est un temps de parcours **linéaire** en n.

`max_somme_lineaire` est donc **un meilleur algorithme (asymptotiquement)** que `max_somme` pour la résolution du problème étudié : le temps de parcours de `max_somme_lineaire` est dominé asymptotiquement par le temps de parcours de `max_somme` : $T''(n) = \mathcal{O}(T'(n))$.

Question : Peut-on mieux faire ?

- Non ! il faut parcourir toute la liste au moins une fois.

Take Home Message

Un algorithme est une série d'étapes permettant de résoudre un problème.

- Il existe plusieurs algorithmes pour résoudre un problème donné, certains pouvant être plus efficaces que d'autres.
- On définit l'efficacité au travers du temps de parcours asymptotique de l'implémentation, en considérant la pire instance possible de ce problème.
- On catégorise ce temps de parcours en utilisant la notation de Landau $T(n) = \Theta(\cdot)$