

Informatique et Calcul Scientifique

Cours 8 : Complexité algorithmique

Critères d'évaluation d'un algorithme

- ▶ Correctitude
- ▶ Performance
 - ▶ Temps de parcours
 - ▶ Espace requis en mémoire
 - ▶ Nombre d'appels à la mémoire de disque
 - ▶ ...

Critères d'évaluation d'un algorithme

- ▶ Correctitude
- ▶ Performance
 - ▶ Temps de parcours
 - ▶ Espace requis en mémoire
 - ▶ Nombre d'appels à la mémoire de disque
 - ▶ ...

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 une liste de n éléments, 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.

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 une liste de n éléments *qui ne grandissent pas avec n* , on dira que l'entrée est de taille n
- ▶ On suppose en général que les opérations suivantes prennent **un temps constant** *si elles s'appliquent à des objets/valeurs qui ne grandissent pas avec n* :
 - ▶ 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

```
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
```

c_0
 c_1
 c_2
 c_4
 c_5
 c_6 } $n-1$ itérations
 c_3

$L = [0, 3, 10, -7, 5]$
 $\text{max_liste}(L)$

L'appel à la fonction `max_liste` dure (en secondes)

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

`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`.

Mesurer le temps de parcours

```
from time import time
from random import randrange

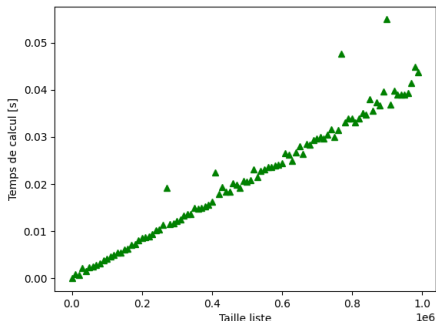
def max_liste(L):
    n = len(L)
    max_L = L[0]
    for i in range(1, n):
        if L[i] > max_L:
            max_L = L[i]
    return max_L

L = []
for i in range(1000000000):
    L.append(randrange(10000))

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


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

Par exemple,

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

Exemple : somme maximale

```
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**.

Correctitude (idée de preuve)

```
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]
```

- ▶ Les boucles imbriquées itèrent sur toutes les paires possibles (i, j) pour i et j distincts. Au début de la " (i, j) ème" exécution du corps de la boucle intérieure, `max_s` contient la valeur de la somme maximale pour toutes les paires vues jusque-là.
- ▶ On peut aussi formuler et prouver un invariant de boucle pour la boucle extérieure, qui dépendra d'un invariant de boucle de la boucle intérieure (qui sera fonction du numéro d'itération i).

Temps de parcours

```
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 ($i = 0, \dots, n-1$).
- ▶ A la i ème itération de la boucle extérieure, la boucle intérieure est exécutée $n - i - 1$ fois :
 - ▶ $i = 0$: j parcourt $\text{range}(1, n)$: $n - 1$ itérations
 - ▶ $i = 1$: j parcourt $\text{range}(2, n)$: $n - 2$ itérations
 - ▶ ...
 - ▶ $i = n-1$: j parcourt $\text{range}(n, n)$: 0 itérations

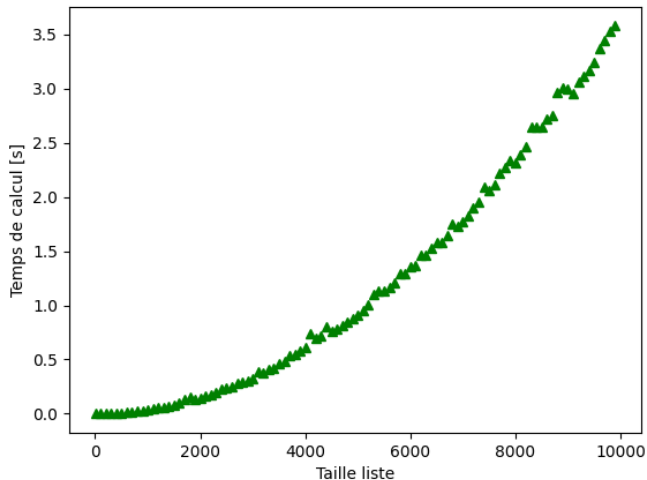
```
for i in range(n):  
    for j in range(i+1, n):  
        if L[i] + L[j] > max_s:  
            max_s = L[i] + L[j]
```

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

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

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

Temps de parcours



Somme maximale : un autre algorithme

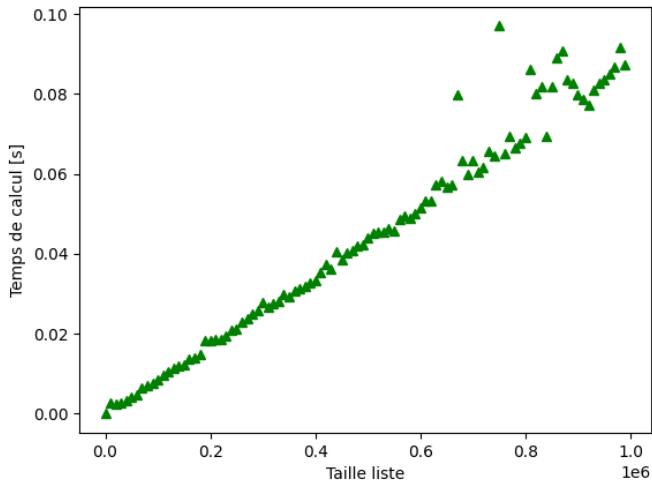
Etant donné une liste de n nombres ($n \geq 2$), donner un algorithme qui calcule la plus grande somme de deux éléments de la liste.

```
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 : $T''(n) = c_3n + c_4$ (si on admet¹ que `L.remove()` a temps de parcours linéaire en n).

1. 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 la liste L.

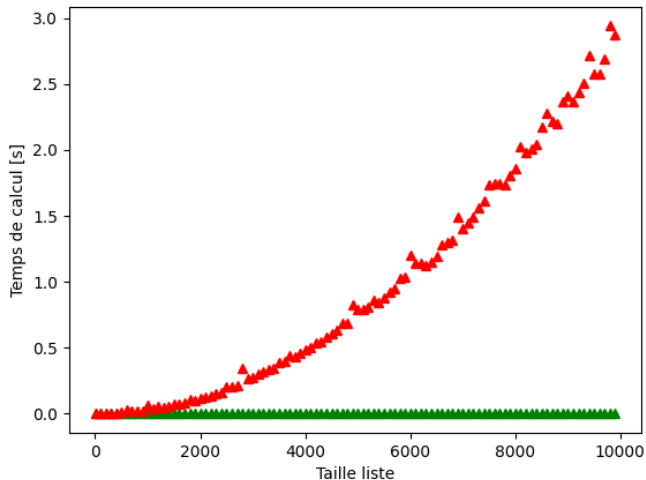
```
from time import time
from random import randrange

L = []
for i in range(1000):
    L.append(randrange(1000))

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



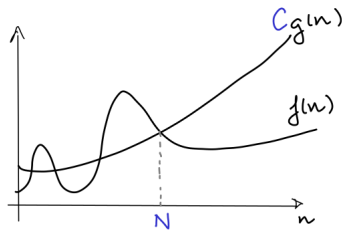
- ▶ Le calcul du temps de parcours avec le module `time` est problématique : le temps de parcours varie d'un langage de programmation à l'autre, d'une machine à l'autre, d'un moment à l'autre sur la même machine...
- ▶ Pour évaluer la performance d'un algorithme indépendamment des détails d'implémentation, 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., au temps de parcours en fonction de la taille de l'entrée n lorsque n tend vers l'infini.

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

Définition (Grand O)

Soit $n \in \mathbb{N}$, 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

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

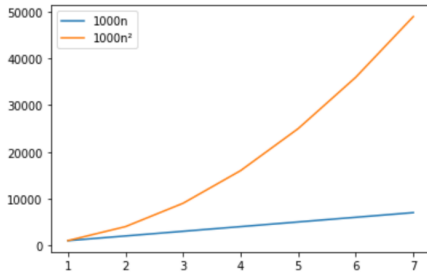


Exemples

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

► $1000n = \mathcal{O}(n^2)$:

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

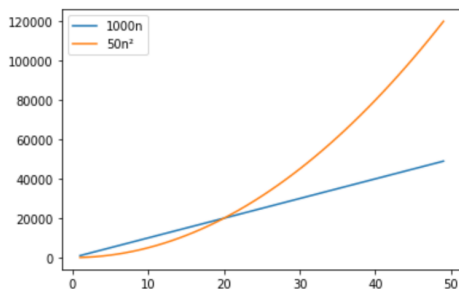


Exemples

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

► $1000n = \mathcal{O}(n^2)$:

$$1000n \leq \underbrace{50}_C n^2 \text{ pour tout } n \geq \underbrace{20}_N.$$

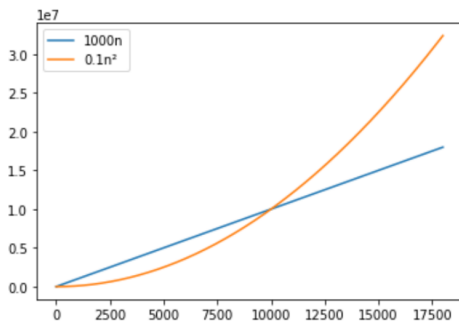


Exemples

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

► $1000n = \mathcal{O}(n^2)$:

$$1000n \leq \underbrace{0.1}_C n^2 \text{ pour tout } n \geq \underbrace{10000}_N.$$



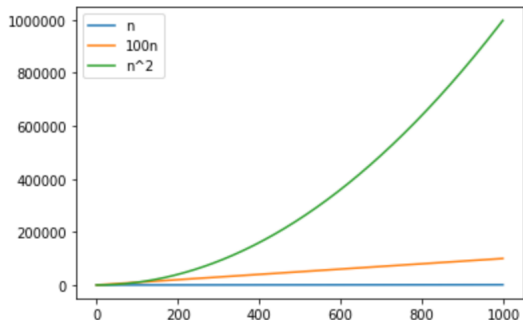
$$f = \mathcal{O}(g) \iff \exists C, N > 0 \text{ t.q. } \forall n > N \ f(n) \leq C \cdot g(n).$$

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

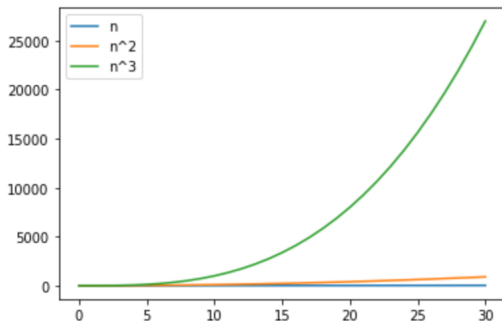
$$n^p = \mathcal{O}(n^q)$$

- ▶ $n = \mathcal{O}(n)$, $n = \mathcal{O}(n^2)$
- ▶ $n^2 = \mathcal{O}(n^3)$
- ▶ $\sqrt{n} = \mathcal{O}(n)$
- ▶ ...
- ▶ Si $p < q$, alors $n^p = \mathcal{O}(n^q)$ mais n^q n'est pas $\mathcal{O}(n^p)$.
- ▶ $n^p = \mathcal{O}(cn^p)$ pour toute constante $c > 0$.

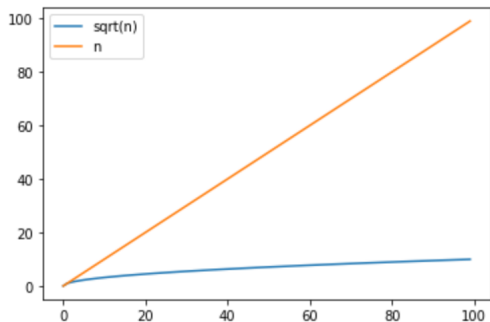
Exemples



Exemples



Exemples



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

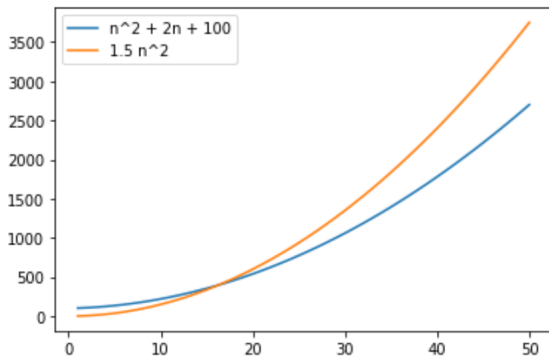
- Pour $p \leq q$ entiers, f , g polynômes de degré p , q respectivement,

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

- $2n + 100 = \mathcal{O}(n^2)$
- $n + 10^6 = \mathcal{O}(n)$
- $n^2 + 1000n + 10^6 = \mathcal{O}(n^3)$
- ...
- Si $p < q$, alors $f = \mathcal{O}(g)$ mais g n'est pas $\mathcal{O}(f)$.

Exemples

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



Sommes de puissances rationnelles

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

- ▶ Plus généralement, pour $p \leq q$ rationnels, f et g des sommes de puissances rationnelles de n dont les plus hautes sont respectivement n^p et n^q ,

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

- ▶ $n + \sqrt{n} = \mathcal{O}(n)$
- ▶ $n\sqrt{n} + 1000n = \mathcal{O}(n^{1.6})$
- ▶ ...
- ▶ Si $p < q$, alors $f = \mathcal{O}(g)$ mais g n'est pas $\mathcal{O}(f)$.

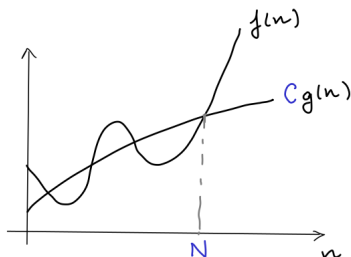
Notation $\Omega(\cdot)$

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 :

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



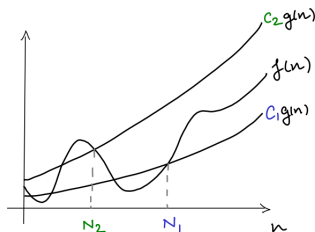
Notation $\Theta(\cdot)$

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 :

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



Exemples

- ▶ Pour tous polynômes f et g de même degré,

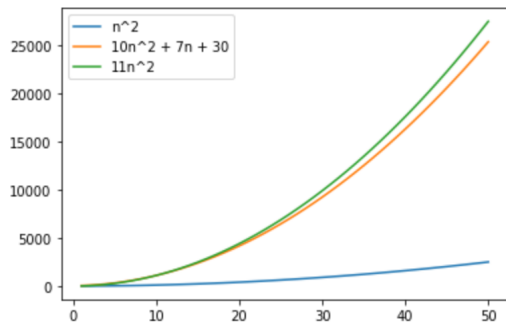
$$f(n) = \Theta(g(n))$$

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

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

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



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
 - ▶ 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 $\Theta(n)$ (linéaire en la taille de l'entrée).
- ▶ Peut-on mieux faire (asymptotiquement) ?

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 $\Theta(n)$ (linéaire en la taille de l'entrée).
- ▶ 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.

Comportement asymptotique des algorithmes de recherche de somme maximale

- ▶ Pour une entrée de taille n :
 - ▶ `max_somme` a temps de parcours (au pire des cas) $T'(n) = c_2 n^2 + c_1 n + c_0$: c'est un temps de parcours **quadratique** en n ($\Theta(n^2)$).
 - ▶ `max_somme_lineaire` a temps de parcours (au pire des cas) $T''(n) = c_3 n + c_4$: c'est un temps de parcours **linéaire** en n ($\Theta(n)$).
- ▶ `max_somme_lineaire` est donc **un meilleur algorithme (asymptotiquement)** que `max_somme` pour la résolution du problème de la somme maximale de deux éléments d'une liste : le temps de parcours de `max_somme_lineaire` est dominé asymptotiquement par le temps de parcours de `max_somme` : $T''(n) = \mathcal{O}(T'(n))$.
- ▶ Peut-on mieux faire?... Non ! borne inférieure triviale : il faut au moins lire toute l'entrée.