Qualité algorithmique

3 — Optimisation de la consommation mémoire

Anthony Labarre

Université Gustave Eiffel

Plan d'aujourd'hui

Les générateurs

Terminologie

- Un **itérable** est un objet capable de renvoyer ses éléments un à un ;
- Un **itérateur** est un objet représentant un flux de données ; il permet d'itérer sur un itérable ;
  - `iter(it)` renvoie un itérateur sur `it`, qu'on fait avancer à l'aide de `next()` ;
  - **`for`** le fait automatiquement ;
- Un **générateur** est une fonction renvoyant un itérateur à l'aide de **`yield`** ;

(source : `https://docs.python.org/3/glossary.html`)

Évaluation fainéante

- Les générateurs (et certains itérateurs) implémentent une **évaluation fainéante** : chaque élément n'est créé qu'au moment où on en a besoin ;

Exemple (évaluation directe vs. fainéante)

```
>>> for x in [elem for elem in liste if elem % 2]:  # évaluation directe
...
>>> for x in filter(lambda elem: elem % 2, liste):  # évaluation fainéante
...
```

- Intérêts :
  1. la consommation en espace mémoire est donc bien moins importante que l'alternative (créer un itérable contenant tous les éléments) ;
  2. on réduit par la même occasion les calculs, puisqu'on évite les coûts liés à l'évaluation directe ;

Écrire ses propres générateurs

Il est très facile d'écrire un générateur : on fait exactement comme pour les fonctions . . . mais on remplace **`return`** par **`yield`**.

Exemple

```
def test():
    yield "Bonjour"
    yield "aux"
    yield "étudiants."


if __name__ == "__main__":
    for valeur in test():
        print(valeur)
```

[Visualisation sur PythonTutor]

Écrire ses propres générateurs

- **`yield`** agit comme **`return`**, mais “se souvient” d’où on en était dans la génération ;
- Cela permet donc d’itérer sur les valeurs renvoyées par **`yield`** au fur et à mesure de leur création, au lieu de devoir renvoyer un itérable les contenant toutes ;

Exemple (générateur de Fibonacci)

La fonction suivante génère les *n* premiers nombres de Fibonacci :

```
def fibo_gen(n):  # utilisation: for x in fibo_gen(n): ...
    """Génère les n premiers nombres de Fibonacci."""
    a, b = 0, 1
    for i in range(n):
        yield a
        a, b = b, a + b
```

Générateurs infinis

On peut pousser les choses encore plus loin et avoir des générateurs infinis !

Exemple (générateur de Fibonacci infini)

```
def fibo_infini():
    """Génère tous les nombres de Fibonacci."""
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b
```

```
if __name__ == "__main__":
    for nombre in fibo_infini():
        print(nombre)
        # prévoir de quoi s'arrêter !
```

Générateurs en compréhension

- Rappelez-vous des compréhensions (de listes, ensembles, dictionnaires) ;
- On peut également définir des générateurs en compréhension, en utilisant simplement des parenthèses ;
- Dans un appel de fonction requérant un itérable, on peut même s'en passer ;

Exemple

```
In [1]: %load_ext memory_profiler
In [2]: %memit
peak memory: 47.73 MiB, increment: 0.22 MiB
In [3]: %memit sum([i for i in range(2 ** 20)])
peak memory: 86.29 MiB, increment: 38.57 MiB
In [4]: %memit sum(i for i in range(2 ** 20))
peak memory: 56.12 MiB, increment: 0.00 MiB
In [5]: %timeit sum([i for i in range(2 ** 20)])
24.6 ms ± 133 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
In [6]: %timeit sum(i for i in range(2 ** 20))
20.1 ms ± 549 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Utilisation

- Les générateurs sont très simples à écrire, mais attention à leur utilisation ;
- Comme les fichiers, on ne peut les parcourir qu'une seule fois !

Exemple

```
>>> intervalle = (2 ** i for i in range(5))
>>> for x in intervalle:
...     print(x, end=" ")
...
1 2 4 8 16
>>> for x in intervalle:
...     print(x, end=" ")
...
# rien
```

- Solution : réinitialiser le générateur (il faut répéter l'affectation) ;
- Remarque : certains itérateurs comme `range` n'ont pas ce problème ;

Utilisation

- L'initialisation d'une variable à l'aide d'un générateur ne déclenche rien !
- On n'obtient les valeurs qu'en appelant `next(objet)` ;

Exemple

```
>>> from fibogen import fibo_gen
>>> nombres = fibo_gen(10)
>>> next(nombres)
0
>>> next(nombres)
1
>>> for n in nombres:
...     print(n)
...
1
2
3
5
8
13
21
34
```

Combinaison de générateurs

L'évaluation fainéante montre toute sa puissance lorsque l'on combine les générateurs ou les itérateurs.

Exemple

Affichons toutes les nombres pairs de la forme k^k pour $k \in \mathbb{N}$:

```
>>> from itertools import filterfalse, starmap
>>> list(filterfalse(lambda x: x % 2, starmap(pow, zip(range(10), range(10)))))
[4, 256, 46656, 16777216]
```

Là encore, chaque nombre du résultat est évalué au fur et à mesure qu'on combine les générateurs (par exemple : `starmap` ne doit pas attendre que `zip` ait tout produit).

Quelques itérateurs utiles (intégrés)

- `enumerate(it)` : renvoie les éléments de `it` numérotés ;
- `filter(f, it)` : renvoie les éléments de `it` pour lesquels `f(x) == True` ;
- `map(f, it)` : renvoie les images des éléments de `it` par `f` ;
- `zip(A, B)` : renvoie (a_0, b_0), (a_1, b_1), … (nécessite $|A| = |B|$) ;

- … et beaucoup d'autres dans le module standard `itertools` ;
- … et encore plus dans le module `more-itertools` (`python3-more-itertools`, `more_itertools`)

Quelques itérateurs utiles (`itertools`)

- `accumulate([1,2,3,4,5])` → `1 3 6 10 15`
- `batched('ABCDEFG', n=3)` → `ABC DEF G`
- `chain('ABC', 'DEF')` → `A B C D E F`
- `chain.from_iterable(['ABC', 'DEF'])` → `A B C D E F`
- `compress('ABCDEF', [1, 0, 1, 0, 1, 1])` → `A C E F`
- `dropwhile(lambda x: x<=5, [1, 4, 6, 4, 1])` → `6 4 1`
- `islice('ABCDEFG', 2, None)` → `C D E F G`
- `pairwise('ABCDEFG')` → `AB BC CD DE EF FG`
- `starmap(pow, [(2, 5), (3, 2), (10, 3)])` → `32 9 1000`
- `takewhile(lambda x: x<=5, [1, 4, 6, 4, 1])` → `1 4`
- `zip_longest('ABCD', 'xy', fillvalue='-')` → `Ax By C- D-`

Coroutines

Coroutines

- Les **coroutines** sont des fonctions procédant du même esprit que les générateurs ;
- La différence est que :
  - les générateurs produisent des éléments un à un ;
  - les coroutines consomment des éléments un à un ;
- Si les données à lire sont trop volumineuses, il peut être utile de pouvoir les communiquer par petits morceaux à une coroutine plutôt qu'en un seul bloc à une fonction traditionnelle ;

Définition d'une coroutine

- On définit les coroutines comme des fonctions classiques ;
- Elles utilisent également le mot-clé **`yield`** . . .
- . . . mais cette fois-ci, pour **affecter** des valeurs au lieu de les renvoyer ;

Exemple

La coroutine suivante réceptionne des nombres, et met à jour leur somme, qu'elle affiche.

```
def total():
    somme = 0
    while True:
        somme += (yield)
        print("La nouvelle somme vaut", somme)
```

Utilisation d'une coroutine

Pour utiliser une coroutine `f`, il faut :

1. l'initialiser (affectation classique via un appel) ;
2. la démarrer (appel à `next()`) ;
3. lui envoyer les données (`send(donnees)`) ;
4. la fermer quand on a fini (`close()`) ;

Exemple

```
>>> def total():
...     somme = 0
...     while True:
...         somme += (yield)
...         print("La nouvelle somme vaut", somme)
...
>>> calculatrice = total()    # initialisation
>>> next(calculatrice)        # démarrage
>>> calculatrice.send(1)      # envoi de données -> exécution du code
La nouvelle somme vaut 1
>>> calculatrice.send(2)      # envoi de données -> exécution du code
La nouvelle somme vaut 3
>>> calculatrice.send(3)      # envoi de données -> exécution du code
La nouvelle somme vaut 6
>>> calculatrice.close()      # on a fini -> on ferme
```

[Visualisation sur PythonTutor]

Coroutines génératrices

On peut utiliser **`yield`** pour recevoir ET renvoyer des données. . . mais il faut être prudent.

Exemple

```
>>> def total():
...     somme = 0
...     while True:
...         somme += (yield)
...         yield somme
...
>>> calculatrice = total()    # initialisation
>>> next(calculatrice)        # démarrage
>>> calculatrice.send(1)      # envoi de données -> exécution du code
1
>>> calculatrice.send(2)      # envoi de données -> exécution du code
# rien ... ???
>>> calculatrice.send(3)      # envoi de données -> exécution du code
4  # euh ...
```

[Visualisation sur PythonTutor]

Explications et solution

- Lorsqu'un **`yield`** s'exécute, il :
  1. renvoie une valeur (ou **`None`**) ;
  2. interrompt la fonction jusqu'à ce qu'on y revienne ;
- `send(x)` fournit `x` au dernier **`yield`** atteint, puis reprend l'exécution de la fonction jusqu'au **`yield`** suivant ;
- Dans notre cas, on récupère le **`None`** implicite du `(yield)` ;
- Solution : appeler `next()` avant **chaque** `send(donnees)`.

Exemple

```
>>> def total():
...     somme = 0
...     while True:
...         somme += (yield)
...         yield somme
...
>>> calculatrice = total()    # initialisation
>>> next(calculatrice)        # démarrage
>>> calculatrice.send(1)      # envoi de données -> exécution du code
1
>>> next(calculatrice)        # avancement
>>> calculatrice.send(2)      # envoi de données -> exécution du code
3
>>> next(calculatrice)        # avancement
>>> calculatrice.send(3)      # envoi de données -> exécution du code
6
```

Décorations pour plus d'ergonomie

- L'utilisation de ces coroutines se complique ;
- Pour plus d'ergonomie, on peut définir des décorateurs qui se chargent automatiquement de rajouter ce qu'on risque d'oublier, à savoir :
  - le `next()` initial ;
  - éventuellement, les `next()` précédant chaque `send()` ;
  - le `close()` final ;

Chaînes “internées”

Internement d'objets

Python optimise gentiment la mémoire à notre place, sans qu'on le lui demande :

Exemple

Examinons la consommation en mémoire d'une liste contenant 10^6 zéros :

```
In [1]: %load_ext memory_profiler
In [2]: %memit ma_liste = [0] * int(1e6)
peak memory: 88.42 MiB, increment: 7.63 MiB
In [3]: set(map(id, ma_liste))
Out[3]: {10861160}
```

- La liste consomme de l'espace à cause de ses nombreuses références à 0 ;
- Mais comme le montre la troisième ligne, toutes ces références pointent vers le même objet en mémoire !
- Ce procédé s'appelle l'**internement** ;

Internement automatique ou manuel ?

L'exemple précédent laisse entendre que l'internement est automatique. Est-ce vrai ?

Exemple

```
>>> premiere = "Voici une chaîne"
>>> deuxieme = "Voici une chaîne"
>>> premiere == deuxieme
True
>>> premiere is deuxieme
False
>>> id(premiere), id(deuxieme)
(140349938451824, 140349938453168)
```

Internement manuel

Si nécessaire, on peut interner soi-même les chaînes que Python “rate”, à l’aide de la fonction `sys.intern(chaine)`.

Exemple

```
>>> from sys import intern
>>> premiere = intern("Voici une chaîne")
>>> deuxieme = intern("Voici une chaîne")
>>> type(premiere), type(deuxieme)
(<class 'str'>, <class 'str'>)
>>> premiere == deuxieme
True
>>> premiere is deuxieme
True
>>> id(premiere), id(deuxieme)
(140453447583088, 140453447583088)
```

Outre la consommation en mémoire réduite, on a également des comparaisons en $O(1)$ (on compare les pointeurs au lieu des contenus).

Structures de données plus économes en mémoire

Structures de données

- Python propose de nombreuses structures de données utiles par défaut ;
- Dans cette partie, on va examiner des alternatives plus économes en mémoire ;
- Elles seront parfois aussi (ou plus) efficaces que les structures que vous connaissez déjà ;
- Le compromis résidera alors dans :
  1. les contraintes d'utilisation ;
  2. la manière de les utiliser ;

Aperçu

On verra des alternatives aux types suivants :

- `list`, `tuple` → tableaux (`array` de divers types) ;
- `set` de naturels → tableaux binaires (`bitarray`) ;
- `set` de chaînes → `Trie` ;

Les tableaux

Le type `array`

- Le type `array` ressemble aux tableaux dynamiques du C(++) ;
- Il existe en fait (au moins) trois types d'`array` en Python, que nous comparerons brièvement :
  - celui du module standard `array` ; (= `array.array`)
  - celui du module `numpy` ; (= `numpy.array`)
  - ceux du module `cyarray` ;

Le type `array.array`

- Le type `array.array` peut s'utiliser comme une liste, mais est plus contraignant :
  1. il ne peut contenir que des nombres ou des caractères (donc des chaînes de longueur 1) ;
  2. on ne peut pas y mélanger les types ;
  3. la taille des éléments est limitée et doit être précisée à l'initialisation ;
- Déclaration :

```
x = array.array(type_elems, iterable);
```

Consommation mémoire de `array.array`

Exemple

Essayons de créer une liste de 100 millions d'entiers, puis un `array.array` contenant les mêmes valeurs :

```
In [1]: %load_ext memory_profiler
In [2]: %memit
peak memory: 72.80 MiB, increment: 0.31 MiB
In [3]: %memit list(range(100000000))
peak memory: 3741.38 MiB, increment: 3668.51 MiB
In [4]: from array import array
In [5]: %memit array("Q", range(100000000))
peak memory: 819.01 MiB, increment: 744.77 MiB
```

Contraintes de `array.array`

Les types utilisables dans un `array.array` sont :

Code d'indication du type	Type C	Type Python	Taille minimum en octets
`'b'`	`signed char`	`int`	1
`'B'`	`unsigned char`	`int`	1
`'u'`	`wchar_t`	Caractère Unicode	2
`'h'`	`signed short`	`int`	2
`'H'`	`unsigned short`	`int`	2
`'i'`	`signed int`	`int`	2
`'I'`	`unsigned int`	`int`	2
`'l'`	`signed long`	`int`	4
`'L'`	`unsigned long`	`int`	4
`'q'`	`signed long long`	`int`	8
`'Q'`	`unsigned long long`	`int`	8
`'f'`	`float`	`float`	4
`'d'`	`double`	`float`	8

Le type de l'`array x` est stocké dans `x.typecode`.

(source : `array`)

Choix du type

Pour optimiser la consommation en mémoire, choisissez le plus petit type permettant de représenter vos données.

Exemple

```
In [1]: %load_ext memory_profiler
In [2]: from array import array
In [3]: %memit array("Q", range(2 ** 26))  # c'est pas mal
peak memory: 551.01 MiB, increment: 478.30 MiB
In [4]: %memit array("I", range(2 ** 26))  # mais on peut faire mieux
peak memory: 327.56 MiB, increment: 254.58 MiB
In [5]: %memit array("H", range(2 ** 26))  # ou se planter en étant radin
...
OverflowError: unsigned short is greater than maximum
```

Le type `numpy.array`

`python3-numpy` `numpy`

- `numpy` propose également son type `array` ;
- Il est moins économe en mémoire que celui du module `array`, mais est parfois plus compact que les listes ;
- On peut mélanger les types, mais ce n'est pas recommandé pour des raisons d'efficacité ;
- On peut utiliser n'importe quel nombre de dimensions ;
- Déclaration :

```
x = numpy.array(iterable);
```

Consommation mémoire de `numpy.array`

Comparons les consommations en mémoire pour la création d'un itérable (`list`, `array.array`, `numpy.array`) des 10^8 premiers naturels :

```
In [1]: %load_ext memory_profiler
In [2]: %memit
peak memory: 72.79 MiB, increment: 0.31 MiB
In [3]: %memit ma_liste = list(range(100000000))
peak memory: 3899.41 MiB, increment: 3826.61 MiB
In [4]: from array import array
In [5]: %memit array("Q", ma_liste)
peak memory: 4641.52 MiB, increment: 741.88 MiB
In [6]: import numpy
In [7]: %memit numpy.array(ma_liste)
peak memory: 4646.00 MiB, increment: 733.91 MiB
```

Remarques sur `numpy`

- Si le but est surtout d'économiser de la mémoire, `numpy.array` ne sera pas nécessairement meilleur que `array.array` (à tester quand même !) ;
- Les fonctions de `numpy` sont plus performantes que celles de Python . . . à condition de ne pas faire de "mélanges" ;

Exemple (`sum` et `list` vs. `numpy.sum` et `numpy.array`)

```
In [1]: import numpy
In [2]: ma_liste = list(range(2 ** 26))
In [3]: mon_array = numpy.array(ma_liste)
In [4]: %timeit sum(ma_liste)
198 ms ± 557 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
In [5]: %timeit numpy.sum(ma_liste)
1.74 s ± 28.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
In [6]: %timeit sum(mon_array)
2.38 s ± 61.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
In [7]: %timeit numpy.sum(mon_array)
26.4 ms ± 81.3 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Les performances sur un `array.array` sont du même ordre.

python3-cyarray cyarray

On peut encore gagner un peu de place avec `cyarray`, si on n'a besoin ni de slices ni d'autre chose que des nombres :

Exemple (test de `cyarray`)

```
In [1]: %load_ext memory_profiler
In [2]: from cyarray.api import IntArray
In [3]: %memit ma_liste = list(range(100000000))
peak memory: 3900.59 MiB, increment: 3823.05 MiB
In [4]: %%memit
   ...: mon_cyarray = IntArray(len(ma_liste))
   ...: for i in range(len(ma_liste)): mon_cyarray[i] = ma_liste[i]
   ...:
   ...:
peak memory: 4294.75 MiB, increment: 381.48 MiB
```

Les types dépendent de la taille des éléments (cf. `array`).

(voir la documentation de `cyarray` pour plus d'infos)

Les tableaux binaires

Les tableaux binaires

- Les **tableaux binaires** sont une implémentation efficace des énumérations (cf. cours de BUT 1) ;
- La position `i` est à `True` si `i` est présent, `False` sinon ;
- Plusieurs packages implémentent cette structure en Python ;

bitarray

python3-bitarray bitarray

Les objets `bitarray` permettent très facilement de représenter un ensemble de naturels.

Exemple

```
In [1]: %load_ext memory_profiler
In [2]: from random import randrange
In [3]: %memit ensemble = {i for i in range(10000000) if randrange(0, 2)}
peak memory: 391.66 MiB, increment: 291.62 MiB
In [4]: from bitarray import bitarray
In [5]: %%memit
...: tableau = bitarray('0') * (max(ensemble) + 1)
...: for elem in ensemble: tableau[elem] = 1
...:
...:
peak memory: 364.04 MiB, increment: 0.00 MiB
In [6]: sorted(ensemble)[:10]
Out[6]: [0, 3, 5, 6, 11, 13, 16, 20, 21, 24]
In [7]: tableau[:10]
Out[7]: bitarray('1001011000')
```

`numpy` propose aussi ce genre de structure, mais en moins ergonomique (voir `packbits` et `unpackbits`).

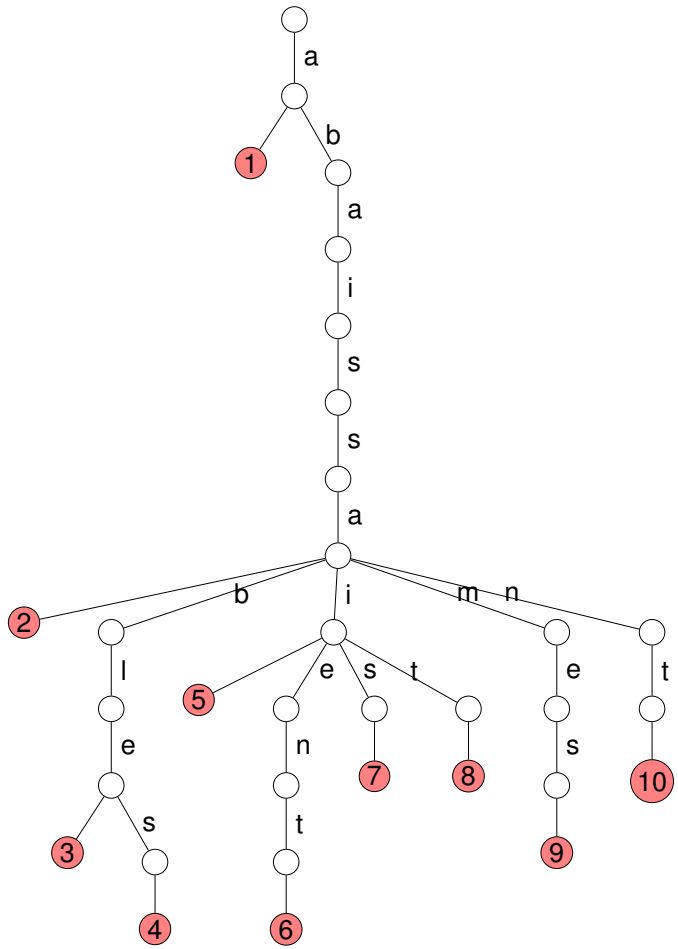
Les tries

Exemple de trie

Les **tries** encodent de manière compacte un ensemble de chaînes, en fusionnant les préfixes communs.

1	a
2	abaissa
3	abaissable
4	abaissables
5	abaissai
6	abaissaient
7	abaissais
8	abaissait
9	abaissames
10	abaissant
11	...

$\Longrightarrow$



Utilisation de `datrie.Trie`

python3-datrie

datrie

Ensemble de chaînes

```
def init_set(chemin):
    resultat = set()
    with open(chemin, "r") as f:
        for ligne in f:
            resultat.add(ligne.strip())

    return resultat
```

```
>>> mes_mots = init_set(...)
>>> "bonjour" in mes_mots
True
>>> "azoinoza" in mes_mots
False
```

```
In [4]: %memit s = init_set("mots.txt")
peak memory: 139.33 MiB,
increment: 37.68 MiB
```

Trie

```
def init_trie(chemin):
    t = datrie.Trie(string.ascii_lowercase)
    with open(chemin, "r") as f:
        for ligne in f:
            t[ligne.strip()] = True

    return t
```

```
>>> mes_mots = init_trie(...)
>>> "bonjour" in mes_mots
True
>>> "bonjo" in mes_mots
False
```

```
In [4]: %memit t = init_trie("mots.txt")
peak memory: 89.12 MiB,
increment: 19.63 MiB
```

Les performances sont un peu moins bonnes :

```
In [10]: %timeit for mot in s: x = (mot in s)
21.7 ms ± 62.9 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
In [11]: %timeit for mot in s: x = (mot in t)
120 ms ± 3.79 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Épilogue

Les tests de régression

Vérifiez toujours que vos optimisations en sont . . . et que le code reste correct !

- Écrivez des tests unitaires pour votre code de départ . . .
- . . . que vous réutiliserez à chaque modification !
- Utilisez les outils adéquats pour ce faire (par exemple `unittest` — oubliez `doctest`) ;

Problèmes d'espace disque

- On a optimisé les performances et la consommation mémoire ;
- On pourrait aussi s'intéresser à l'espace disque ;
- Si vous manquez d'espace disque, vous pouvez comprimer vos fichiers . . .
- . . . et les manipuler directement dans votre programme ;
- Plusieurs modules standards permettent de gérer les formats les plus fréquents :
  - `bz2`
  - `gzip`
  - `lzma`
  - `tarfile`
  - `zipfile`
  - `zlib`

Exploitation des cartes graphiques (GPUs)

- Nous nous sommes limités aux processeurs classiques (CPU) et à leurs cœurs pour les calculs ;
- Les cartes graphiques (GPU) contiennent également des processeurs ;
- On pourrait donc s'intéresser à utiliser en parallèle non seulement les cœurs du CPU, mais également ceux du GPU ;

GPUs : Parallélisme extrême

- Les CPUs actuels contiennent au mieux plusieurs dizaines de cœurs (Intel en propose 24) ;
- Sur les GPUs, on les compte par centaines… voire par milliers (Nvidia GeForce RTX 4090: 16 384) ;
- `multiprocessing` ne suffit plus, on a besoin de modules spécialisés ; par exemple :
  - `pycuda` (`python3-pycuda`, `pycuda`) ;
  - `pytorch` (`python3-torch`, `pytorch`) ;
- Voir Gorelick et Ozsvald, pages 189–197 ;

Structures de données plus efficaces

Certains algorithmes et structures de données plus performantes vues dans d'autres cours d'algorithmique sont disponibles en Python :

- `collections.deque` : insertions et suppressions en $O(1)$ en tête et en fin ;
- `heapq` : files à priorité (extraction du minimum en $O(\log n)$) ;
- `bisect` : maintien de listes triées ;

Introspection

- L'**introspection** est la capacité d'un programme à examiner son propre état ;
- Elle est possible en Python à l'aide des modules standards `inspect` et `traceback` ;
- Elle nous ouvre pas mal de portes :
  - détecter si une fonction est récursive ;
  - examiner le code d'une fonction pour repérer les types qui y sont utilisés ;
  - …
- … que l'on peut exploiter à l'aide de décorateurs ;

Introspection : exemples

`inspect` permet (entre autres) d'obtenir :

- le code source d'un objet fonction (non intégrée)
  `getsource(f), getsourcelines(f)`
- les noms et valeurs des paramètres d'un appel
  `getcallargs(f, *args, **kwargs)`
- le contenu d'un module
  `getmembers(module_importé)`
- le type précis d'un objet (fonction, générateur, ...)
  `isfunction(obj), isgenerator(obj), ...`
- les contextes des appels de fonction, voire toute la pile
  `currentframe(), stack()`

`traceback` permet (entre autres) :

- d'intercepter les exceptions tout en récupérant le message d'erreur
  `format_exc()`

Idées d'application

Quelques idées de décorateurs utiles basés sur `inspect` :

- suggérer automatiquement l'utilisation de :
  - `lru_cache` pour les fonctions récursives ;
  - "meilleures" structures de données ;
  - …
- empêcher l'exécution du code selon certaines conditions sur des paramètres ;
- réécrire du code à la volée ;

⋮