0 Quelques rappels et compléments

I Rappels d'algorithmique

1 Preuve d'algorithme

On rappelle que la preuve d'un algorithme passe par :

- **Une preuve de terminaison** pour les boucles conditionnelles (`while`). Cela se fait via un *variant de boucle* : en général une suite entière minorée strictement décroissante.
- **La détermination d'*invariants de boucle*** : propriétés décrivant l'état du système (ou une partie de celui-ci) à un instant donné d'une boucle, qui se prouve par récurrence.
- **La correction de l'algorithme** : on peut alors connaître l'état du système en fin d'exécution et donc vérifier que la quantité renvoyée est la bonne, ou que le traitement effectué est le bon.

Exemples 1 : Exponentiation naïve et exponentiation rapide itérative

E1 – Écrire une fonction `puiss(x, n)` renvoyant x^n et prouver sa correction.

E2 – Pour calculer x^n, on remarque la chose suivante $x^{2^{k+1}} = \left(x^{2^k}\right)^2$: il suffit donc d'une multiplication pour passer de x^{2^k} à $x^{2^{k+1}}$.

L'idée est alors de décomposer n en base 2 : si n s'écrit

$$n = b_0 + b_1 \cdot 2 + b_2 2^2 + \cdots + b_k 2^k$$

avec $b_0, \cdots, b_k \in \{0,1\}$ (écriture en base 2) alors

$$x^n = x^{b_0} \left(x^2\right)^{b_1} \cdots \left(x^{2^k}\right)^{b_k},$$

le calcul de x^{2^p} s'effectuant à partir de celui de $x^{2^{p-1}}$.

Par exemple, $19 = 1+2+2^4$ et $x^{19} = x \times x^2 \times \left(\left(\left(x^2\right)^2\right)^2\right)^2$ (6 multiplications nécessaires), $39 = 1+2+2^2+2^5$ et

$$x^{39} = x \times x^2 \times \left(x^2\right)^2 \times \left(\left(\left(\left(x^2\right)^2\right)^2\right)^2\right)^2$$

(8 multiplications nécessaires).

Écrire une fonction `exp_rap(x, n)` mettant on œuvre cet algorithme et prouver sa correction.

E3 – Écrire enfin une version récursive `exp_rap_rec(x, n)` basée sur le fait que

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ \left(x^{n/2}\right)^2 & \text{si } n \text{ est pair} \\ x \cdot \left(x^{\lfloor n/2 \rfloor}\right)^2 & \text{si } n \text{ est impair} \end{cases}$$

et prouver sa correction.

2 Calculs de complexité

a Principe

Calculer la complexité d'un algorithme revient à évaluer le temps d'exécution et/ou la place occupée dans la mémoire pendant l'exécution.

La mesure peut se faire de manière empirique (`time()`, `timeit()`...) mais dépendra de la machine, d'autres tâches en cours, de l'échantillon de test, etc. ou de manière théorique, mais cela demande de modéliser la machine et de choisir ce qui va être compter, certaines opérations pouvant être plus pertinentes que d'autres (plus lentes ou pour comparer plusieurs algorithmes. Pour calculer la complexité d'un tri d'un tableau, par exemple, on peut choisir de compter plutôt les comparaison entre éléments du tableau ou bien les échanges d'éléments dans le tableau (ou les deux).

Les opérations élémentaires prennent, suivant les machines et les langages, entre 1 ns (10^{-9} s) et 1 μs (10^{-6} s) à être exécutées :

- addition, soustraction, multiplication, division, modulo sur des entiers ou des flottants, comparaison de nombres,
- affectation simple,
- accès aux éléments d'un tableau,
- modification des éléments d'un tableau,
- taille d'un tableau,
- La méthode `append` sur une liste Python peut être considérée comme une opération élémentaire.

b Notations

Notation 1 : O

On suppose que $f(n) \geqslant 0$ et $g(n) \geqslant 0$ pour tout entier n.
On note $f = O(g)$ lorsqu'il existe un rang n_0 et une constante $c > 0$ tels que

$$\forall\, n \geqslant n_0,\quad f(n) \leqslant c \times g(n)$$

Cela signifie que f croît au plus aussi vite que g.

Remarque 1

Les calculs de complexité en moyenne étant souvent compliqués à mener, faisant intervenir des probabilités, le programme se limite aux complexités au mieux ou au pire. En général, une complexité s'exprimera $T(n) = O(f(n))$ avec $f(n)$ optimisé.

c Tableau comparatif

Chaque valeur est multipliée par 10^{-6} s comme majorant du temps d'exécution d'une opération élémentaire.

Croissance	n	10	50	100	500	1000
logarithmique	$\ln n$	2 μs	4 μs	4,6 μs	6 μs	7 μs
	$\sqrt{n}$	3 μs	7 μs	10 μs	20 μs	30 μs
linéaire	n	10 μs	50 μs	100 μs	0,5 ms	1 ms
semi-linéaire	$n\ln n$	20 μs	200 μs	500 μs	3 ms	7 ms
quadratique	n^2	100 μs	2,5 ms	10 ms	0,25 s	1s
polynomiale	n^3	1 ms	0,1 s	1 s	2 min	16 min
exponentielle	2^n	1 ms	36 a	$4\cdot10^6$ a	10^{137} a	$3\cdot10^{287}$ a
	$n!$	4 s	10^{51} a	$3\cdot10^{144}$ a	$3\cdot10^{1121}$ a	$3\cdot10^{2554}$ a

Ordres de grandeurs :

- **Âge de l'univers** : $13,8 \times 10^9$ années
- **Nombre d'atomes dans l'univers** : 10^{80}.

Échelle de croissances comparées :

$$\ln n \ll \sqrt{n} \ll n \ll n\ln n \ll n^2 \ll n^3 \ll 2^n \ll n! \ll n^n$$

n maximum pour un temps d'exécution d'une seconde :

$\ln n$	$\sqrt{n}$	n	$n\ln n$	n^2	n^3	2^n	$n!$
$\simeq 10^{400\ 000}$	10^{12}	10^6	87848	10^3	100	20	10

On retiendra qu'à partir de n^3, le temps d'exécution n'est pas raisonnable.

Exercice 1

Calculer les complexités des fonctions d'exponentiation précédentes.

II Structures de données abstraites

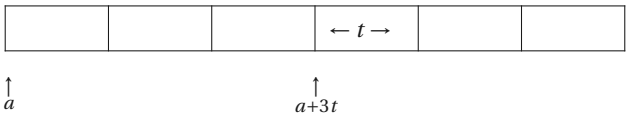
Il convient dans la pratique de l'algorithmique de choisir de représenter ses données en utilisant une structure appropriée, dépendant de ce qu'l'on a besoin d'en faire et de ce que cela coûte.

On rencontre deux types de structures : celles pour lesquelles un emplacement dans la mémoire (RAM) aura été alloué dès la création et ne changera pas, dites **statiques** et celles dont l'allocation mémoire et la taille peuvent varier au cours de l'exécution d'un algorithme, dites **dynamiques**. Lorsque l'on peut modifier les éléments de cette structure, elle dite **mutable**.

Par exemple, en python, les tuples sont statiques, non mutables, et les listes sont dynamiques et mutables.

1 Tableau

Un tableau informatique est une structure statique pour laquelle on fixe à la création un nombre d'éléments et des espaces mémoires contigu pour chaque élément (de type fixé). Connaissant l'adresse a de premier élément et la taille t correspondant au type d'élément, on accède à l'élément numéro k à l'adresse $a+kt$.

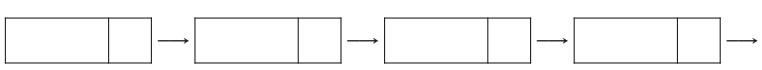


- **Avantages :** chaque élément est accessible en lecture et en écriture à temps constant, ainsi que la taille du tableau.
- **Inconvénients :** structure statique : taille non modifiable. Insertion, suppression lentes.

Le module `numpy` fournit un type `ndarray` correspondant à un tableau informatique.

2 Liste chaînée

Une liste est une structure dynamique qui est telle que chaque élément à stocker est encapsulé dans un objet contenant aussi un pointeur vers l'élément suivant. Pour atteindre l'élément numéro k, on est obligé de parcourir tous les éléments précédents dans la liste.



- **Avantages :** Structure dynamique, insertion, suppression rapides.
- **Inconvénients :** k^{e} élément accessible en $O(k)$.

Il existe d'autres types de listes (doublement-chaînée : chaque élément pointe vers son successeur et son prédécesseur, circulaire : le dernier élément pointe sur le premier, etc.)

⚠ Comme son nom ne l'indique pas, le type `list` de python n'est pas une liste chaînée. C'est un ingénieux mélange de tableau et de liste. Voir partie suivante.

3 Les piles

Le principe de la pile informatique (*stack* en anglais) est résumé par l'acronyme LIFO :

L*ast* **I***n* **F***irst* **O***ut*

C'est le principe d'une pile d'assiettes :

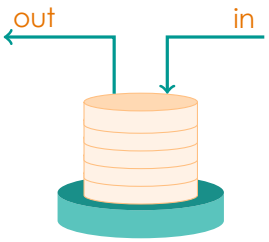
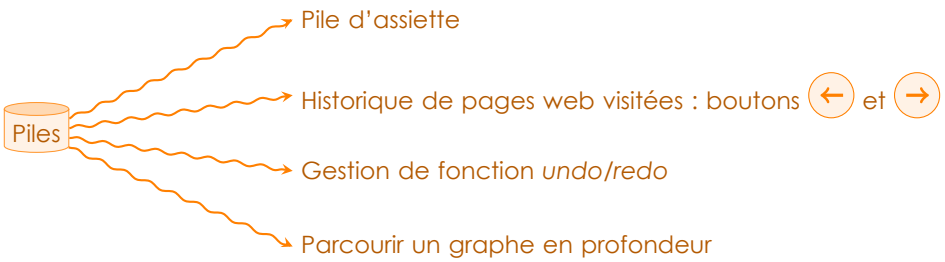


Figure 1 – Structure de pile (LIFO)

Voici quelques exemples d'usage de cette structure :



Pour implémenter une structure de pile, on a besoin d'un nombre réduit d'opérations de bases :

- Créer une pile vide,
- Tester si une pile est vide,
- Dépiler le dernier élément du haut (**pop**) à temps constant,
- Empiler un élément (**push**) à temps constant.

À cela peuvent éventuellement s'ajouter des fonctions permettant de :

- renvoyer la taille de la pile,
- renvoyer le sommet de la pile sans la modifier (= pop + push).

4 Les files

Il existe également une structure FIFO (**F***irst* **I***n* **F***irst* **O***ut*) de file (*queue* en anglais).

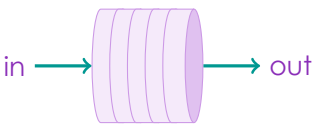
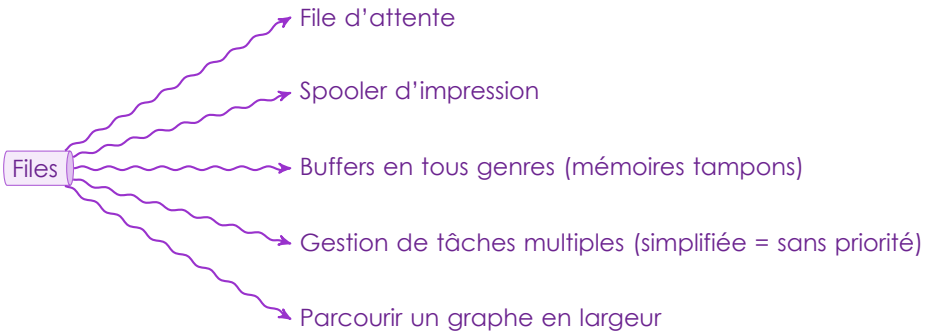


Figure 2 – Structure de file (FIFO)

Voici quelques exemples d'usage de cette structure :



Pour implémenter une structure de file, on a besoin d'un nombre réduit d'opérations de bases :
- Créer une file vide,
- Tester si une file est vide,
- Défiler le dernier élément du haut (**enqueue**) à temps constant,
- Enfiler un élément (**dequeue**) à temps constant.

À cela peuvent éventuellement s'ajouter des fonctions permettant de :
- renvoyer la taille de la file,
- renvoyer le premier élément de la file sans la modifier.

III Python

Après un an d'utilisation régulière de Python, vous devez être à l'aise avec ce langage. Attention à ne pas négliger la présentation du code pour optimiser la lisibilité : docstring et PEP 8 sont de rigueur.

1 Les listes Python

Python permet de travailler avec des objets de type `list` qui est une structure de données dynamique mutable : ce n'est cependant ni un tableau ni une liste au sens informatique du terme.

Le principe est le suivant : lorsque l'on crée une liste ou lorsque l'on ajoute des éléments à une liste, on réserve une place plus grande en mémoire. Plus exactement, la liste est « allongée » lorsque l'on passe certains paliers qui sont :

$$0,4,8,16,25,35,46,58,72,88,106,126,148,173,201,233,\ldots$$

D'où viennent ces valeurs ? Pour le savoir, il faut fouiller de le code source de CPython, consultable librement :

```
/* This over-allocates proportional to the list size, making room
* for additional growth.  The over-allocation is mild, but is
* enough to give linear-time amortized behavior over a long
* sequence of appends() in the presence of a poorly-performing
* system realloc().
* The growth pattern is:  0, 4, 8, 16, 25, 35, 46, 58, 72, 88, ...
*/
new_allocated = (newsize >> 3) + (newsize < 9 ? 3 : 6);

/* check for integer overflow */
if (new_allocated > PY_SIZE_MAX - newsize) {
    PyErr_NoMemory();
    return -1;
} else {
    new_allocated += newsize;
}
```

Le `>> 3` signifie un décalage de 3 bits vers la droite, donc une division entière par 2^3 et `newsize < 9 ? 3 : 6` vaut 3 si `newsize < 9` et 6 sinon.

Autrement dit, si n est la taille de la liste, la place allouée dans la mémoire sera

$$n+\left\lfloor\frac{n}{8}\right\rfloor+\begin{cases}3 & \text{si } n<9\\ 6 & \text{sinon}\end{cases}$$

Voir `https://www.laurentluce.com/posts/python-list-implementation/` pour la gestion détaillée de l'allocation de mémoire lors de la manipulation de liste.

Python

```
>>> L = [1, 2, 3]
>>> L[2]
3

>>> L[1] = 0 # Mutable
>>> L
[1, 0, 3]
>>> L[1:3] # Slicing
[0, 3]

>>> len(L)
3

>>> L + [1, 2, 3] # Concatén.
[1, 0, 3, 1, 2, 3]

>>> 3 in L # Appartenance
True
>>> tuple(L) # Casting
(1, 0, 3)
>>> list(t)
[1, 2.65, True, 'azerty']
>>> str(L)
'[1, 0, 3]'
>>> list("Bonjour")
['B', 'o', 'n', 'j', 'o', 'u', 'r']
# Ajouter un élément à la fin
>>> L.append(5)
>>> L
[1, 0, 3, 5]

# Récupérer le dernier élément
>>> a = L.pop()
>>> a
5
>>> L
[1, 0, 3]
```

Python

```
>>> L2 = L
>>> id(L), id(L2)
(140352491149648, 140352491149648)
>>> id(L) == id(L2)
True
>>> L[1] = 12
>>> L
[1, 12, 0]
>>> L2
[1, 12, 0]

>>> L3 = L.copy()
# ou L[:] ou list(L)
>>> id(L) == id(L3)
False
>>> L[1] = 36
>>> L
[1, 36, 0]
>>> L3
[1, 12, 0]

>>> LL = [[1],[2],[3]]
>>> LL2 = LL.copy()
# copie superficielle
>>> LL2
[[1], [2], [3]]

>>> LL2[0][0] = 0
>>> LL2
[[0], [2], [3]]
>>> LL
[[0], [2], [3]]

>>> LL2[0] = [1]
>>> LL
[[0], [2], [3]]
>>> LL2
[[1], [2], [3]]
```

- ⚠ Certaines opérations peuvent se révéler dangereuses : on évitera à tout prix d'écrire, par exemple

  `L = [a] * n`

  mais on préférera `L = [a for _ in range(n)]`.
  Si `a` n'est pas mutable, cela ne pose pas de problème, mais s'il l'est, c'est très embêtant ! Le même genre de problème peut se présenter lorsque l'on utilise `+=` avec des listes de listes.

- Notons que le module `copy` fournit une fonction `deepcopy` permettant de faire des copies profondes de listes.

- On rappelle également que les listes sont passées **par référence** lorsqu'elles sont arguments de fonctions : cela signifie que la liste peut être modifiée à l'intérieur d'une fonction dont elle est l'un des arguments.

- Lu sur `https://wiki.python.org/moin/TimeComplexity` :

  Generally, 'n' is the number of elements currently in the container. 'k' is either the value of a parameter or the number of elements in the parameter. (1) = These operations rely on the "Amortized" part of "Amortized Worst Case". Individual actions may take surprisingly long, depending on the history of the container.

Operation	Average Case	Amortized Worst Case
Copy	$O(n)$	$O(n)$
Append[1]	$O(1)$	$O(1)$
Insert	$O(n)$	$O(n)$
Get Item	$O(1)$	$O(1)$
Set Item	$O(1)$	$O(1)$
Delete Item	$O(n)$	$O(n)$
Iteration	$O(n)$	$O(n)$
Get Slice	$O(k)$	$O(k)$
Del Slice	$O(n)$	$O(n)$
Set Slice	$O(k+n)$	$O(k+n)$
Extend[1]	$O(k)$	$O(k)$
Sort	$O(n \log n)$	$O(n \log n)$
Multiply	$O(nk)$	$O(nk)$
x in s	$O(n)$	
min(s), max(s)	$O(n)$	
Get Length	$O(1)$	$O(1)$

2 Les piles

Simuler une pile est très simple avec python : il suffit d'utiliser une liste.

- La création d'une liste vide : `pile = []`,
- Le test de liste vide : `pile == []`,
- L'ajout d'un élément avec la méthode : `pile.append(x)`
- L'extraction du dernier élément avec la méthode : `x = pile.pop()` **sans argument**

se font à coût constant.

3 Les files

Pour simuler une file, utiliser une liste python n'est pas une bonne idée. L'utilisation de `L.pop(0)` pour défiler est bien trop coûteuse.

On utilise les listes doublement chaînée `deque` du module `collections` qui, en plus de proposer des méthodes `append(x)`, `pop()` s'exécutant à coût constant comme pour les listes, propose des méthodes `appendleft(x)`, `popleft()` permettant d'insérer un élément au début ou d'extraire le premier élément à coût constant.

On peut donc simuler une file, en ayant appelé au préalable

```
from collections import deque
```

- La création d'une file vide : `file = deque()`,
- Le test de liste vide : `file == deque()`,
- L'ajout d'un élément avec la méthode : `file.append(x)`
- L'extraction du dernier élément avec la méthode : `x = file.popleft()` **sans argument**

se font à coût constant.

Bien sûr, les `deque` du module `collections` peuvent aussi servir à simuler des piles.

4 Les fichiers

Ce paragraphe a pour seul but d'introduire le mot-clé `with` que l'on s'efforcera d'utiliser pour la gestion de flux : ouverture de fichier, de base de donnée, etc.

Par exemple :

Python
```
with open(nomFichier, 'r') as fichier:
    <...>
    <instructions utilisant fichier>
    <...>
```

Cela permet d'optimiser la gestion de la mémoire en fermant le fichier automatiquement dès que le traitement est terminé.

Il faut savoir utiliser les méthodes `read`, `write`, `readline`, `readlines` sur les fichiers.

Retenir également qu'on peut itérer sur un fichier : `for line in file:...`