Chapitre III

Programmation d'algorithmes de base

Table des matières

Partie A

Recherche séquentielle dans un tableau unidimensionnel. Dictionnaire

Dans cette partie, nous nous intéresserons à la programmation d'algorithmes de recherches dans un tableau, par exemple, la recherche d'un élément donné ; d'un maximum lorsque les éléments font partie d'ensemble ordonné. Nous utiliserons également la structure de dictionnaire pour compter les occurrences des éléments d'un tableau.

1. Tableau unidimensionnel

En algorithmique, les tableaux unidimensionnel s'apparentent aux suites mathématiques : ils permettent de manipuler sous un même nom plusieurs variables (a priori) de même type chacune repérée par un numéro : son indice dans le tableau.

a. Définitions

Définition 1. *Tableau unidimensionnel*

Un **tableau unidimensionnel** est une collection finie d'éléments dont chacun est désigné de manière unique par un nombre entier naturel appelé **indice** de l'élément dans le tableau. *Le temps d'accès aux éléments d'un tableau par leurs indices est constant.*
On appelle **taille** d'un tableau la nombre d'éléments qui le compose.

Exemple 1. *Tableaux en Python*

Dans la suite, nous programmerons nos tableaux en Python grâce au type `list` ou encore `tuple`. Nous avons vu rapidement le type `array` fourni par le module `numpy` ; ce type est un "meilleur" représentant de ce qu'est un tableau en informatique : données de même type et ajout/suppression impossibles pour garantir un accès aux éléments en temps constant.

Tableau avec le type `list`

```
>>>t = [1,2,7,5,3]
```

Tableau avec le type `tuple`

```
>>>t = (1,2,7,5,3)
```

Dans la suite de cette partie et dans les T.P. qui en dépendent, nous pourrons appliquer nos programmes de recherche indépedemment aux types `list` et `tuple` car la syntaxe d'accès aux élements est la même pour ces deux types.

Remarque 1.

Dans certains langages de programmation (les langages à typage statique comme C ou OCaml), les éléments d'un tableau doivent tous avoir le même type, on définit dans ce cas le **type d'un tableau** comme étant le type commun de ses élements.
En Python, qui est un langage à typage dynamique, les listes (type `list`) et les tuples (type `tuple`) permettent des collections hétérogènes d'élements, par exemple, `[1,'coucou',[2,3]]` est valide.

2. TP : Recherche séquentielle dans un tableau unidimensionnel

Dans le T.P. suivant, nous allons étudier le principe de recherches séquentielles dans un tableau :

TP n°2 - Recherches séquentielles

Partie B

Fonctions récursives

Dans cette partie, nous allons étudier une autre façon de penser nos algorithmes. Jusqu'à présent, nous avons utilisé des boucles (for ou while) pour les programmer - on dit que ce sont des algorithmes *itératifs* ; le principe de *récursivité* va nous permettre d'avoir une nouvelle approche, dans certains cas plus naturelle, pour le faire.

1. Définition et premier exemple

Définition 2. *Fonction récursive*

On appelle **fonction récursive** est une fonction qui s'appelle elle-même.

Pernons l'exemple du calcul de $n!$ pour un entier naturel n :

Remarque 2. *Version itérative*

Nous connaissons déjà l'algorithme itératif permettant ce calcul, celui-ci étant basé sur l'égalité : pour $n \in \mathbb{N}$,

$$n! = \prod_{k=1}^{n} k$$

```python
def facto_it(n):
    p=1
    for k in range(1,n+1):
        p=p*k
    return k
```

Exemple 2. *Version récursive*

L'algorithme récursive est basé sur la définition *récurrente* de $n!$:

$$\begin{cases} 0! = 1 \\ n! = n \times (n-1)! \quad \text{pour } n \in \mathbb{N}^* \end{cases}$$

On remarque alors que si on connait la valeur de $(n-1)!$, on obtient la valeur de $n!$ en la multipliant simplement par n. Et si on ne connait pas $(n-1)!$, on l'obtient en multipliant $(n-2)!$ par $n-1$; et si on ne connait pas $(n-2)!$ ... etc...
A force de "faire -1", on arrivera sur $0!$ dont on connait la valeur. Il ne reste plus qu'à remonter toute cette chaîne pour obtenir $n!$.

Voyons comment on peut traduire algorithmiquement ce principe :

```
1 def facto_rec(n):
2     if n==0:
3         return 1
4     else:
5         return n*facto_rec(n-1)
```

Pour mieux comprendre le déroulement d'un appel de la fonction `facto_rec`, on ajoute y les instructions suivantes qui nous permettront de suivre ce qui s'y passe :

Regardons sous le capot !

```
1 def facto_rec(n):
2     if n==0:
3         print('\nappel de fact_rec('+str(n)+') : renvoie 1\n')
4         return 1
5     else:
6         print(n*'--'+' appel de fact_rec('+str(n)+')')
7         f=facto_rec(n-1)
8         p = n*f
9         print(n*'--'+' fin de fact_rec('+str(n)+') :',end=' ')
10        print('calcule et renvoie p='+str(n)+'x'+str(f)+'='+str(p))
11        return p
```

Ainsi on obtient le résultat suivant lors de l'appel de `facto_rec(5)` :

```
>>> facto_rec(5)
---------- appel de fact_rec(5)
-------- appel de fact_rec(4)
------ appel de fact_rec(3)
---- appel de fact_rec(2)
-- appel de fact_rec(1)

appel de fact_rec(0) : renvoie 1

-- fin de fact_rec(1) : calcule et renvoie p=1x1=1
---- fin de fact_rec(2) : calcule et renvoie p=2x1=2
------ fin de fact_rec(3) : calcule et renvoie p=3x2=6
-------- fin de fact_rec(4) : calcule et renvoie p=4x6=24
---------- fin de fact_rec(5) : calcule et renvoie p=5x24=120
120
```

On se rend alors compte que tant que `facto_rec(0)` n'est pas appelé, aucun calcul n'est effectué. Et ensuite, une fois l'appel de `facto_rec(0)` effectué, les résolutions de chaque appel de `facto_rec(k)` se font dans l'ordre inverse des appels initiaux.
Métaphoriquement, le "fonctionnement" est celui d'une pile d'assiettes que l'on doit laver : on empile les assiettes sales (la première arrivée est en dessous, la dernière au dessus) puis lorsqu'on commence le nettoyage, on lave donc en premier la dernière arrivée, etc... et on lave en dernier la première arrivée !

Ainsi, pour qu'un langage puisse réaliser l'implémentation de la récursivité, on a besoin d'utiliser une structure de données particulière qui puisse "empiler" et "dépiler" les appels successifs d'une fonction récursive : il s'agit de la structure de ... *pile* ! Comme la pile d'assiettes donc !
Nous ne nous attarderons pas sur cette les détails de cette structure mais il s'agit ici d'un des principaux désavantage de la programmation récursive par rapport à la programmation itérative : la pile garde en mémoire les appels, il faut donc stocker cette pile... qui peut prendre beaucoup de place !
Par exemple, Python limite par défaut la pile à 1000 appels récursifs :

```
>>>facto_rec(1000)
RecursionError: maximum recursion depth exceeded while calling a Python object
```

Et ceci afin d'éviter une saturation de la mémoire vive.
On peut bien-sûr changer la limite imposée par défaut en utilisant le module `sys` :

```
>>>import sys
>>>sys.setrecursionlimit(5000) # pile limitée à 5000 appels
>>>facto_rec(1026)
5698395808375297261747844963126775835219334225754751838890410520163998440186966505315 9
    5771596685007012311962854089849609872581664432217545831106589320408677263314710834
    6243056979768248254377332444669615657353805287933132804990745743436490546903976669
    3886370575102284196941111949287562466469997622850389992065498828621531983932769995
    7206943604720056348555251114997127184044626152798950846310539707249930379111416307
    6793609334457154029752202864690187585028113670367330335224091281023732663206758416
    5507646190128304947980906160645175847671648989797918951360039497163417733417591559
    3826797383249739127776431318783919051945752640927694153887809539186135702431437116
    6426774578603524594693116716521970423082341950766707575399960572618428805272146207
    6935452184361597896230964088688481505646602321434053678540730095576754002130336447
    6610780068095098585644945125247169438400768959744789337458294231391905037101213485
    2250751744638147302359794838301600116661696031020532611008465136962727004241241737
    4982865010718000175174934504786652679088913137796904568310573945317096559675694347
    8491861885945035702303577566065843747422049972228299796668304255621879443966776850
    9703495970689058430536113081036405602976042301869812952259153413715727931939751856
    5650583094394949834126589250713288365950879126494671198325157758513400449625383867
    2851987803347747632245674403324908887107173386037805623943094563784848087174119598
    2519174135291963492577509253811313588031220068654769079277242788884830432304572276
    1199615851791323398223151320092904210686101300936648508706279086950159235524551270
    9557827167148204289326352096695831273794314100488748206466780008617409165536518994
    9345015724046045514393607362688302414067574210017983036565908127331851070187305567
    4151643295614599747674979049361715811625057444782113173761095884275543228492283115
    9647657823229774066427585080280404994816221249246572304726285798369894523022857765
    9241735867925765207655041991429850436773763870427583078181211018991021098519202263
    8350469537878657346173737357305111136886767095806758457895003297367717814706191955
    6855163806595855416732912162345057349683389543738906034936592196794441053393718071
    3201929335748388505691507902137911337780729141374170550968312289963345835867397074
    3740126468026425825066597247661725226132333749701191004608097161552234561792397161
    4362425615191731103332570751985099649414129445168980050236224618642815408879904485
    2441415680000000000000000000000000000000000000000000000000000000000000000000000
    0000000000000000000000000000000000000000000000000000000000000000000000000000000
    0000000000000000000000000000000000000000000000000000000000000000000000000000000
    00000000000000000
```

2. TP : Récursivité

Dans le T.P. suivant, nous allons nous exercer à programmer récursivement :

TP n°5 - Récursivité

Partie C

Algorithmes dichotomiques

Dans cette partie, nous allons étudier des algorithmes **dichothomique** : le principe de ces algorithmes repose sur la "devise" : *diviser pour régner* qui est un principe de base en algorithmie. L'idée de résolution d'un problème par un algorithme dichotomique est de couper le problème en deux sous-problèmes que l'on résout de nouveau par ce même algorithme ! Voici quelques exemples de ce principe qui semble de prime abord déroutant :

1. Recherche dichotomique

Dans la première partie de ce chapitre et les T.P. associés, nous avons étudié et programmé un algotithme de recherche d'un élément dans un tableau dont les données n'avaient pas de propriété particulière. Nous avons vu que dans le pire des cas - l'élément recherché n'est pas dans le tableau - l'algorithme faisait autant de comparaison que la taille du tableau ce qu'on a traduit par une complexité temporelle linéaire de l'algorithme.
En supposant une propriété supplémentaire sur les données du tableau, nous allons voir qu'on peut diminuer drastiquement le nombre de comparaison et donc la complexité dans un nouvel algorithme de recherche : la recherche dichotomique.

Principe de la recherche dichotomique :
On considère un tableau `L` que l'on suppose **triée** on suppose donc que ses éléments appartiennent à un ensemble muni d'un (pré-)ordre total et on considère un élément `x` que l'on se propose de chercher dans `L`.

- Si `L` est vide, `x` n'est pas dans `L`.
- Si `L` n'est pas vide, on "coupe" le tableau `L` en deux en son milieu (ou presque) et on considère l'élément `l` à cet endroit :
  - si `l` vaut `x`, alors `x` est dans `L` ;
  - si `l` est strictement plus petit que `x`, alors, `L` étant triée, `x` est potentiellement dans la moitié droite de `L` ;
  - si `l` est strictement plus grand que `x`, alors, `L` étant triée, `x` est potentiellement dans la moitié gauche de `L` ;
- Dans les deux derniers cas, on répète l'algorithme sur la moitié qui contient potentiellement `x`.

Décrite de la sorte, on "sent" que la recherche dichotomique se prète parfaitement au principe récursif. Dans le T.P. qui suivra, nous programmerons l'algorithme tout de même de manière itérative et récurive pour comparer ces deux philosophies.

2. Exponentiation rapide

Voici un deuxième exemple d'algorithme dichotomique : l'algorithme d'**exponentiation rapide** qui permet de calculer un nombre à une puissance entière. Son principe est le suivant :

Principe de l'exponentiation rapide :
On considère un nombre x et un entier naturel n.

- Si n vaut 0, on renvoie 1.
- Si $n \geq 1$:
  - si n est pair, on a $x^n = \left(x^{\frac{n}{2}}\right)^2$;
  - si n est impair, on a $x^n = x\left(x^{\frac{n-1}{2}}\right)^2$;
- puis on répète l'algorithme pour le calcul de $x^{\frac{n}{2}}$.

Comme pour la recherche dichotomique, la description précédente de l'exponentiation rapide est récursive ; mais nous la programmation selon les principes itératif et récursif dans le T.P. suivant.

3. TP : Algorithmes dichotomiques

Dans le T.P. suivant, nous allons nous programmer la recherche dichotomique et l'exponentiation rapide :

TP n°7 - Algorithmes dichotomiques