Chapitre 2

Étude d'algorithmes et Dictionnaires

Table des matières

1 Étude théorique d'algorithme

L'informatique est une science, à ce titre il ne s'agit pas seulement de taper des lignes de codes et de constater que l'on a le résultat attendu. Il y a toute une partie dont le but est aussi d'étudier ce qui a été fait. On peut se poser trois questions :

- Est-ce que mon code s'arrête bien ? En effet, on sait que l'on peut avoir des boucles `while` infinies [1]. Dans des codes simples, cela peut paraître évident qu'il n'y a pas de problème, mais dans des problèmes plus compliqués. On va devoir étudier la question. Ce problème s'appelle le **problème de terminaison**.
- Lorsque le code termine bien, est-ce que le résultat est bien le résultat voulu ? Cette question s'appelle le **problème de correction partielle**.
- Si la correction est partielle et qu'en plus on a montré la terminaison, alors on dit que la **correction est totale**.
- Enfin si le code s'arrête bien, et que le résultat est celui voulu, combien de temps/combien d'opérations ont été nécessaires ? Il est évident qu'entre un algorithme qui donne le bon résultat en une heure et un autre en deux mois on préferera le premier, surtout si le but de l'algorithme est de prédire le temps qu'il fera demain. Ce problème s'appelle le **problème de complexité**.

Opérations	Syntaxe	Complexité
affectation	**a=2**	$\mathcal{O}(1)$
sur les entiers	+,-,*,//,%	$\mathcal{O}(1)$
sur les flottants	+,-,*	$\mathcal{O}(1)$
comparaison	>=, <=, ==, >=	$\mathcal{O}(1)$
Ajout d'un élément à une liste	`L.append(3)`	$\mathcal{O}(1)$
Rechercher si un élément est dans une liste (hors programme)	`x in L`	$\mathcal{O}(n)$

TABLE 1 – Complexité de quelques opérations

2 Les tris

2.1 Le tri fusion

2.1.1 Le code du tri fusion

Nous ne reviendrons pas sur l'explication du tri fusion dans ce document. Nous allons redonner le code et l'analyser.

```
def Fusion(G,D):
    """Fusionne deux listes triées en une liste triée"""
    i,j = 0,0
    N=[]
    n,p = len(G),len(D)
    while i < n and j < p:
        if G[i] < D[j]:
            N.append(G[i])
            i = i + 1
        else:
            N.append(D[j])
            j = j + 1
    for k in range(i,n):
        N.append(G[k])
    for k in range(j,p):
        N.append(D[k])
    return N
```

1. On peut aussi avoir une boucle `for` infinie.

```python
def TriFusion(L):
    """Pour une liste L, renvoie une liste triée contenant les éléments de L"""
    if len(L) <= 1:#Une liste vide ou à un élément est déjà triée
        return L
    else:
        G=L[0:len(L)//2]#Moitié gauche de L
        D=L[len(L)//2:len(L)]#Moitié droite de L
        G=TriFusion(G)#On trie G par récursivité
        D=TriFusion(D)#On trie D par récursivité
        F=Fusion(G,D)#On fusionne de façon à ce que ça reste trié
        return F#On renvoie le résultat
```

2.1.2 La terminaison du tri fusion

Montrons la terminaison de cet algorithme.

Tout d'abord étudions la fonction `Fusion` : dans la première boucle while i+j est un variant de boucle car augmente de 1 à chaque fois, or i+j<n+p donc la première boucle while s'arrête. La deuxième s'arrête car i est aussi un variant de boucle et augment de 1 à chaque itération de même pour la troisième boucle while.

Pour montrer la terminaison de `TriFusion`, on peut faire une récurrence sur la longeur de `L`, si len(L) <= 1, alors la terminaison est bien vérifiée. Soit $n \in \mathbb{N}^*$, supposons que `TriFusion` termine bien pour toute liste de longueur inférieure ou égale à n. Considérons une liste de longeur $n+1$, alors `G` et `D` seront deux listes de longueur inférieur ou égale à n et donc `TriFusion(G)` et `TriFusion(D)` termineront bien. Enfin, la `Fusion(G,D)` finira bien d'après ce qui précède.

2.2 Complexité du tri fusion

Pour calculer la complexité, on va nommer $C(n)$ le nombre d'opérations qu'il faut faire pour trier une liste de longueur n dans le pire des cas.

Comme on coupe à chaque fois des listes en deux. On va partir d'une liste `L` à 2^n éléments (ainsi quand on la coupe en deux, on obtient deux listes avec 2^{n-1} éléments). Tout d'abord il faut créer les listes `G` et `D`. Ceci prend 2^{n-1} opérations pour `G` et autant pour `D`, soit 2^n opérations. Ensuite, il faut trier `G` comme `G` est une liste à 2^{n-1} éléments, la complexité est de $C(2^{n-1})$ de même pour `D`, ainsi, on obtient

$$C(2^n) = 2^n + C(2^{n-1}) + C(2^{n-1}) + R$$

où R est la complexité dans le pire des cas pour exécuter `Fusion(G,D)`. On observe que dans la fonction `Fusion(G,D)` les éléments de `G` et de `D` vont devoir tôt ou tard être rajouté à la liste `N` au cours d'une des trois boucles `while` et à chaque fois il y a un ajout et une variable qui augment de plus 1. Il y a aussi dans la première boucle `while` une comparaison. Il y a donc $3n$ opérations dans le pire des cas.

Soit donc une complexité de $R = 3 \times 2^n$. Dès lors, on obtient :

$$C(2^n) = 2C(2^{n-1}) + 4 \times 2^n$$

Malheureusement, on obtient une suite qui n'est pas une suite arithmétique, ni géométrique, ni arithmético-géométrique. L'astuce consiste à diviser par 2^n. On obtient alors :

$$\frac{C(2^n)}{2^n} = \frac{C(2^{n-1})}{2^{n-1}} + 4$$

Ainsi, posons $u_n = C(2^n)/2^n$, on obtient alors que $u_n = u_{n-1} + 4$. Dès lors, $(u_n)_n$ est une suite arithmétique de raison 4. Donc $u_n = u_0 + 4n$ D'où $C(2^n) = 2^n(4n + u_0)$. Comme $u_0 = C(1) = 1$ (un seul test à faire pour une liste de longueur 1). Ainsi, $C(2^n) = 2^n(4n + 1)$. Dès lors, en notant, $m = 2^n$, alors $n = \log_2(m)$, on obtient

$$C(m) = m(4\log_2(m) + 1) = \mathcal{O}(m\log_2(m))$$

Mais seulement pour les listes dont le nombre d'éléments m est une puissance de deux.

Considérons $k \in \mathbb{N}$ quelconque et cherchons à encadrer $C(k)$. Il existe $n \in \mathbb{N}$ tel que $2^n \leqslant k < 2^{n+1}$. En effet :

$$2^n \leqslant k < 2^{n+1} \quad \Longleftrightarrow \quad n \leqslant \log_2(k) < n+1 \quad \Longleftrightarrow \quad n = E(\log_2(k))$$

Il suffit donc de prendre $n = E(\log_2(k))$ où E désigne la partie entière et $\log_2$ le logarithme en base 2.

En outre C est une fonction croissante [2] Donc, $C(2^n) \leqslant C(k) \leqslant C(2^{n+1})$. Donc,

$$2^n(4n+1) \leqslant C(k) \leqslant 2^{n+1}(4n+5)$$

En utilisant que $\log_2(k) - 1 < n \leqslant \log_2(k)$, on obtient que

$$2^{\log_2(k)-1}(4(\log_2(k)-1)+1) \leqslant C(k) \leqslant 2^{\log_2(k)+1}(4\log_2(k)+5)$$

On observe alors que

$$2^{\log_2(k)+1}(4\log_2(k)+5) = 2k(4\log_2(k)+5) = \mathcal{O}(k\log_2(k))$$

Ceci démontre que $C(k) = \mathcal{O}(k\log_2(k))$ que k soit une puissance de 2 ou non.

La complexité du tri fusion de longueur n est donc un $\mathcal{O}(n\log_2(n))$. Cette complexité est appelée **quasi-linéaire**. Ceci qui est bien mieux qu'une complexité quadratique comme le tri à bulles. Cela explique pourquoi le tri fusion est effectif pour des listes à 100000 éléments, là où le tri à bulles ne semble pas finir.

2.3 Correction du tri fusion

3 Assert, spécification

`assert` est un mot clé que l'on fait suivre par une condition. Si cette condition n'est pas vérifiée, alors le programme renvoie une erreur en indiquant que c'est telle **assertion** qui n'a pas été vérifie. Supposons que l'on fasse un programme qui ait besoin de diviser par un nombre, alors : warning !

```
def Inverse(x):
    assert x!=0
    return 1/x
```

4 Jeu de tests

Un jeu de test permet de vérifier si le programme donne bien les bonnes réponses

5 Compléments sur les listes

5.1 Boucle sur les éléments d'une liste

Au lieu de boucler sur les indices d'une ligne, on peut boucler sur les éléments d'une liste. Par exemple :

```
def Maximum(L):
    assert len(L)>0
    M=L[0]
    for i in range(len(L)):#i est un indice et L[i] est l'élément de L d'indice i
        if L[i]>M:
            M=L[i]
    return M

def Maximum(L):
    assert len(L)>0
    M=L[0]
    for e in L:#m est un élément de la liste L
        if e>M:
            M=e
    return M
```

2. En effet, intuitivement, plus une liste a d'éléments plus il faudra d'étapes pour la trier. Ce n'est pas un argument très formalisé mais c'est l'idée.

5.2 Liste par compréhension

Étant donnée une liste, on peut créer une autre liste ayant une certaine propriété. Par exemple, si `L=[0,2,4,0,-2,3,4]`, alors on peut définir :

```
M=[e for e in L if e!=0]
N=[e for e in L if e%2==0]
O=[e**2 for e in L if e!=0]
```

On peut aussi définir les listes avec des `append` mais cela prend plus de lignes, par exemple :

```
M=[]
for e in L:
    if e!=0:
        M.append(e)
```

Ainsi, si on veut une liste contenant que des 0 de longeur p, on écrit :

```
L=[0 for j in range(p)]
```

Si on veut une matrice à n lignes et p colonnes, alors on se rappelle qu'il faut donc une liste de listes ou chaque liste représente une ligne, ainsi chaque ligne est de la forme `[0 for j in range(p)]`, comme on veut n lignes, on crée une liste ou cette ligne est répétée n fois soit :

```
M=[[0 for j in range(p)] for i in range(n)]
```

6 Dictionnaires

Lorsque l'on a une liste, on accède aux éléments de cette liste grâce aux indices qui sont des entiers. Plus précisément, les indices possibles sont les éléments de $[\![0\,;n-1]\!]$ si n est le nombre d'éléments de la liste. Ainsi, si `L=[5,"a",["c",3.2]]`, alors `L[0]` vaut 5, `L[1]` vaut `"a"` et `L[2]` vaut `["c",3.2]`. Évidemment la commande `print(L[4])` provoquera une erreur : `out of range`.

On aimerait généraliser le concept de listes, en nous permettant d'avoir des indices qui ne serait pas forcément des entiers entre 0 et $n-1$. Et c'est précisément là où les dictionnaires interviennent. La notion d'indices une liste va être remplacée par celles de clés.

Donnons un exemple, imaginons qu'on veuille compter le nombre de pommes de poires et de fraises. On pourrait enregistrer par une liste : `L=[5,2,25]` et on a qu'a retenir que `L[0]` va compter le nombre de pommes, `L[1]` le nombre de poires et `L[2]` le nombre de fraises. Mais ce n'est pas commode, imaginez que la liste est un million d'éléments, vous n'avez pas très envie d'apprendre par cœur que `L[987123]` compte le nombre d'arbouses que vous avez.

Un dictionnaire (ou tableau associatif) est une collection de paires de la forme `(clé,valeur)`, ainsi les différents fruits en votre possessions seront vos clés et les valeurs seront le nombre pour chaque fruit.

6.1 Créer un dictionnaire

Il existe trois façons de créer un dictionnaire.

1. Créer un dictionnaire vide : `D={}`.
2. Créer un dictionnaire par extension `D={"pommes":5,"poires":2,"fraises":25}`.

6.2 Vérifier si un élément est une clé dans un dictionnaire

Supposons que l'on ait un dictionnaire `D={"pommes":5,"poires":2,"fraises":25}` La commande `"pommes" in D` est un booléen qui renverra `True` si "pommes" est une clé de D et `False` sinon. Ainsi `"pommes" in D` renverra `True` tandis que `5 in D` et `"kakis" in D` renverront `False`.

Remarque 1. Vérifier si un élément est dans une liste a une complexité en $\mathcal{O}(n)$ si n est la longueur de la liste (en effet, une telle vérification consiste à passer en revu chaque élément de la liste). Tandis que vérifier si un élément est dans un dictionnaire a une complexité en $\mathcal{O}(1)$ (ce miracle sera expliqué l'année prochaine).

6.3 Afficher la valeur d'un dictionnaire avec une clé

La commande `print(D["pommes"])` affichera le nombre de pommes.

6.4 Modifier les valeurs d'un dictionnaire

La commande `D["pommes"]=10` modifie la valeur de `D` pour la clé `"pommes"`

6.5 Rajouter une clé et une valeur à un dictionnaire

Vous venez de recevoir 10 kakis, alors indiquez-le avec la commande `D["kakis"]=10`.

6.6 Accéder aux clés/valeurs d'un dictionnaire

Les commandes `D.keys()`, `D.values()` et `D.items()` renvoient respectivement la collection des clés, des valeurs et des couples (clé, valeurs), attention ce ne sont pas des listes, on peut les convertir en liste si besoin grâce aux commandes `list(D.keys())`, `list(D.values())` et `list(D.items())`. On peut aussi boucler sur les clés, valeurs ou items d'un dictionnaire :

```
for cle in D.keys():#for cle in D: fait la même chose en plus court
  print(cle)
  print(D[cle])

for valeur in D.values():
  print(valeur)

for item in D.items():
  print(item)
```

À noter que `len(D)` donne aussi le nombre de couples (clé,valeur) dans le dictionnaire.

6.7 Copie d'un dictionnaire

On rappelle que si `L` est une liste alors la commande `M = L` ne créera pas une liste distincte de `L` toute modification de `L` ou de `M` affectera l'autre liste. Il faut avoir en tête le résultat des commandes suivantes :

```
L = [1,2,3]
M = L
M[2] = 5
```

Pour parer à ce problème, on pourra utiliser la commande `M=L.copy()`. Ainsi, le code suivant donnera bien le résultat attendu :

```
L = [1,2,3]
M = L.copy()
M[2] = 5
```

Pour les dictionnaires, c'est le même problème, le code suivant ne donne pas le résultat escompté :

```
D = {"a":1,"b":2,"c":3}
Dp = D
D["b"] = 5
```

Par contre, la commande `copy` fonctionne de la même manière que pour les listes :

```
D = {"a":1,"b":2,"c":3}
Dp = D.copy()
Dp["b"] = 5
```

Attention, la commande copy() ne fonctionne pas si les éléments du dictionnaire ou de la liste sont eux-mêmes des dictionnaire ou des listes.

6.8 Un exercice classique à savoir faire absolument

Écrire une fonction qui à, une liste, renvoie un dictionnaire comptant le nombre d'occurrences de chaque élément de la liste :

```
def CompterOccurences(L):
  """renvoie un dictionnaire D dont les clés sont les éléments de L
  tel que D[x] contient le nombre d'occurrences de x dans L"""
  D = {}
  for i in range(len(L)):
    if L[i] in D:
        D[L[i]] = D[L[i]] + 1
    else:
        D[L[i]] = 1
  return D
```

On peut faire le même programme mais en bouclant sur les éléments de la liste plutôt que sur les indices :

```
def CompterOccurences(L):
  """renvoie un dictionnaire D dont les clés sont les éléments de L
  tel que D[x] contient le nombre d'occurences de x dans L"""
  D = {}
  for e in L:
    if e in D:
        D[e] = D[e] + 1
    else:
        D[e] = 1
  return D
```