Plan du chapitre

1 Plusieurs algorithmes pour traiter un même problème

1.A Position du problème

Pour traiter un même problème, il existe souvent plusieurs algorithmes. Quand on doit choisir, le critère principal est le temps d'exécution.

Un logiciel interactif par exemple exige un temps de réponse court pour le confort de l'utilisateur.

De même, de nombreux programmes industriels doivent être utilisés un grand nombre de fois dans un délai très court.

Notez aussi qu'un programme exécuté une seule fois, comme par exemple un programme de simulation qui teste une hypothèse de recherche, est inutilisable s'il demande des mois ou des années de calcul !

1.B Un exemple introductif : calcul d'une valeur approchée du nombre π

Les techniques du programme de mathématiques de deuxième année permettent de montrer que :

$$\pi = \lim_{n\to+\infty} (u_0 + u_1 + u_2 + \ldots + u_n)$$

où, $(u_n)_{n\in\mathbb{N}}$ est la suite définie par récurrence par :

$$u_0 = 2 \qquad \text{et} \qquad \forall n \in \mathbb{N}^*,\ u_n = \frac{(2n-1)^2}{2n(2n+1)} u_{n-1}.$$

1.B.1 Une première fonction

On se propose d'écrire une fonction `sommeDesU` qui prend un entier naturel n comme argument et qui renvoie la somme $u_0 + u_1 + \ldots + u_n$. Une première méthode consiste à utiliser une fonction `u` prenant comme argument un entier naturel $k \geq 0$ et qui renvoie la valeur de u_k.

```
1 def u(k):
2     A=2  #A vaut u0
3     for i in range(1,k+1):
4         #A vaut u(i-1)
5         A=A*(2*i-1)**2/2/i/(2*i+1)
6         #A vaut u(i)
7     return A
```

L'écriture de la fonction `sommeDesU` est alors élémentaire :

```
1 def sommeDesU(n):
2     sommeur=2
3     for k in range(1,n+1):
4         sommeur=sommeur+u(k)
5     return sommeur
```

On peut mesurer le temps d'exécution de l'appel `sommeDesU(n)` avec la fonction `time()` du module `time` (qui renvoie le nombre de secondes écoulées depuis le 1er janvier 1970 à 00 : 00 : 00) :

```
t=time.time()
A=sommeDesU(1000)
print("Temps exécution:",time.time()-t)
```

```
Temps exécution: 18.28739903
```

L'algorithme précédent est peu efficace, car il réalise beaucoup de multiplications/divisions superflues. L'appel de `sommeDesU(n)` va provoquer les appels de `u(1)`,`u(2)`,...,`u(n)`. Sachant que l'appel `u(k)` coûte $7k$ multiplications/divisions, l'appel de `sommeDesU(n)` réalise donc :

$$7\sum_{k=1}^{n} k = \frac{7n(n+1)}{2} \text{ multiplications/divisions.}$$

1.B.2 Une deuxième fonction

Un algorithme plus efficace consiste à modifier la fonction `sommeDesU` de sorte qu'elle ne fasse plus appel à `u` ; on utilise alors une variable `U`, qui vaut presque à chaque instant u_k :

```
def sommeDesURapide(n):
    U=2# U contient u(0)
    sommeur=2
    for k in range(1,n+1):
        #U contient u(k-1)
        U=U*(2*k-1)**2/2/k/(2*k+1)
        #U contient u(k)
        sommeur=sommeur+U
        #sommeur contient u(0)+...+u(k)
    return sommeur
```

L'appel de `sommeDesURapide(n)` réalise seulement $7n$ multiplications/divisions, soit $\frac{n+1}{2}$ fois moins que `sommeDesU(n)`, ce qui divise approximativement le temps d'exécution par $\frac{n}{2}$.

Par exemple, pour $n = 5000$, le temps d'exécution est approximativement 2500 fois plus court :

```
t=time.time()
A=SommeDesURapide(5000)
print("Temps exécution:",time.time()-t)
```

```
Temps exécution:0.00721976
```

Le tableau suivant présente les temps d'exécution pour différentes valeurs de n :

	$n = 500$	$n = 1000$	$n = 2000$
Nombre de mul./div. pour `sommeDesU(n)`	876750	3503500	14007000
Temps d'exe. de `sommeDesU(n)`	0.16777027	0.68486203	2.85468780
Nombre de mul./div. pour `sommeDesURapide(n)`	3500	7000	14000
Temps d'exe. de `sommeDesURapide(n)`	0.00069139	0.00150918	0.00309590

On peut remarquer que, quand la valeur de n double, le temps d'exécution de `sommeDesU(n)` est approximativement multiplié par 4, tandis que celui de `sommeDesURapide(n)` est approximativement multiplié par 2.

Cela s'explique aisément : le nombre de multiplications/divisions effectuées par `sommeDesU(n)` est $\frac{7n^2}{2} + \frac{7n}{2}$, qui, pour les grandes valeurs de n, est équivalent à $\frac{7n^2}{2}$ ($\frac{7n}{2}$ qui augmente beaucoup moins vite, est négligeable devant $\frac{7n^2}{2}$). Or $\frac{7n^2}{2}$ est multiplié par 4 quand n double.

Traçons maintenant les courbes des temps d'exécution en fonction de n :

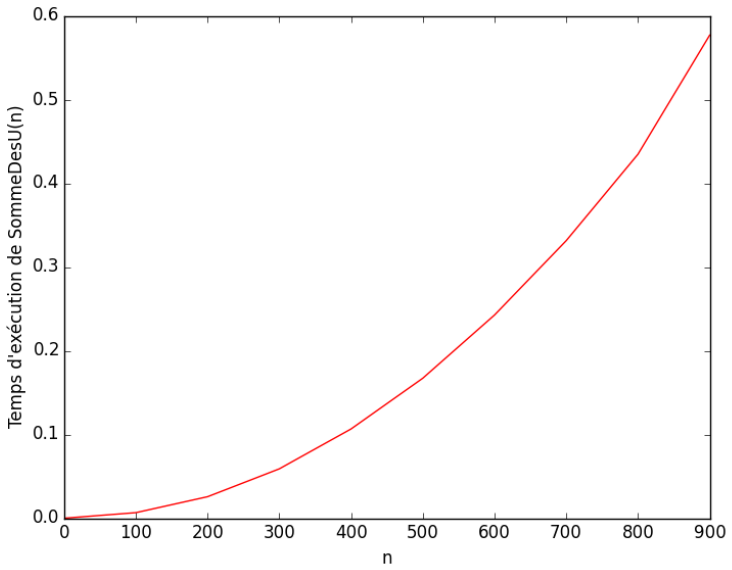

FIGURE 1 – Temps d'exécution de `sommeDesU`

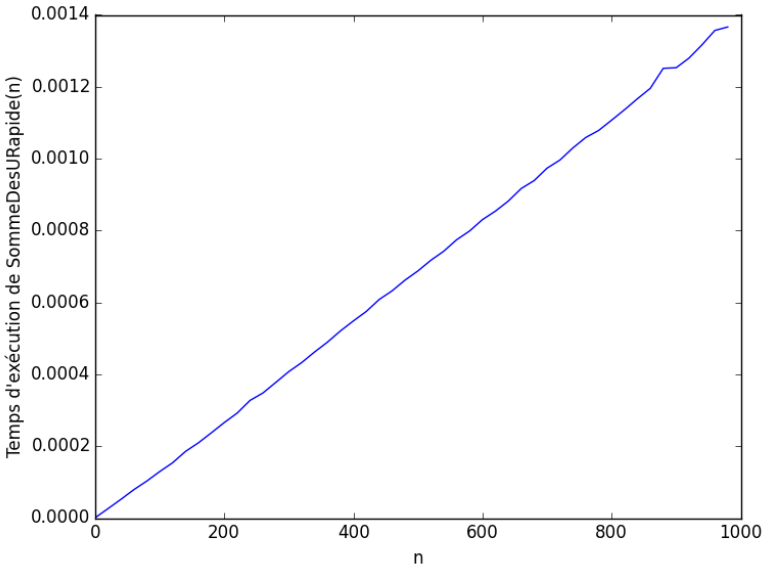

FIGURE 2 – Temps d'exécution de `sommeDesURapide`

On remarque que la courbe du temps d'exécution de `sommeDesU` a l'allure d'une parabole, tandis que celle du temps d'exécution de `sommeDesURapide` ressemble à une droite.

Dans la suite, nous dirons que la complexité de `sommeDesU` est **quadratique**, tandis que celle de `sommeDesURapide` est **linéaire**.

1.C Savoir déterminer le coût d'un algorithme

Dans un premier temps, on cherche une grandeur n pour « mesurer la taille » des données traitées. On calcule les performances (ou le nombre d'opérations) uniquement en fonction de n. Dans la pratique,

- ▷ si la donnée est un entier naturel n, on prend n comme taille du problème ;
- ▷ si la donnée est un couple (p, q) de deux entiers naturels, on prendra suivant le contexte $n = \max(p, q)$, $n = p \times q$, ou $n = p + q$;
- ▷ si la donnée est une liste, on prendra pour n son nombre d'éléments ;
- ▷ si la donnée est une chaîne de caractères, on prendra pour n son nombre de caractères.

On peut distinguer trois catégories d'opérations : les affectations, les comparaisons et les opérations algébriques sur les nombres ($+$, $\times$, division).

Le contexte va nous indiquer ce qu'il convient de prendre en compte. Par exemple, pour chercher un élément dans une liste, on ne compte que le nombre de comparaisons ; pour calculer le terme d'indice n d'une suite définie par récurrence, on ne compte que les opérations algébriques sur les nombres.

Remarque pratique : comment évaluer le coût d'une boucle

- ▷ Le coût d'une boucle `for` du type :

```
for i in iterable:
    p
```

est égal au nombre d'éléments de l'itérable multiplié par le coût de l'instruction `p` si ce dernier ne dépend pas de la valeur de `i`.

- ▷ Quand le coût du corps de la boucle dépend de la valeur de `i`, le coût total de la boucle est la somme des coûts du corps de la boucle pour chaque valeur de `i`.

Exercice 1

On considère la suite $(u_n)_{n\in\mathbb{N}}$ définie par :

$$u_0 = 1 \qquad \text{et} \qquad \forall n \in \mathbb{N}, u_{n+1} = \frac{1}{1+u_n^2}.$$

Ecrire une fonction qui retourne u_n, avec n en argument.
Mesurer sa complexité en fonction de n en comptant le nombre d'opérations algébriques effectuées.

```
def suite(n):
    u=1 # u vaut u(0)
    for k in range(1,n+1):
            #u vaut u(k-1)
        u=1/(1+u**2)
            #u vaut u(k)
    return u    #u vaut u(n)
```

Lors de chaque itération de la boucle, 3 opérations algébriques sont effectuées.
Sachant qu'il y a en tout n itérations de la boucle, l'appel `suite(n)` réalise $3n$ opérations.

Exercice 2

Mesurer la complexité de la fonction suivante en comptant le nombre d'opérations effectuées en fonction de n.

```
def f1(n):
    x=0
    for i in range(n):
        for j in range(n):
            x+=1
    return x
```

Exercice 3

Mesurer la complexité de la fonction suivante en comptant le nombre d'opérations effectuées en fonction de n.

```
1 def f2(n):
2     x=0
3     for i in range(n):
4         for j in range(i):
5             x+=1
6     return x
```

▷ Le cas des boucles `while` est plus complexe à traiter puisque le nombre de répétitions n'est en général pas connu *a priori*. On peut majorer le nombre de répétitions de la boucle et ainsi majorer le coût de l'exécution de la boucle.

Exercice 4

Mesurer la complexité de la fonction suivante en comptant le nombre d'opérations effectuées en fonction de n.

```
1 def f3(n):
2     x, i = 0, n
3     while i > 1:
4         x += 1
5         i //= 2
6     return x
```

1.D Quelques exemples de calcul de complexité

Exercice 5

Ecrire une fonction qui renvoie la moyenne d'une liste de flottants passée en entrée. Evaluer sa complexité.

```
1 def moyenne(l):
2     s=0
3     for x in l:
4         s=s+x
5     return (s/len(l))
```

Si n désigne la longueur de `l`, le nombre d'opérations algébriques réalisées par l'appel `moyenne(l)` est $n+1$

Exercice 6

Comparer les complexités des trois fonctions suivantes :

```
1 def fonction1(n):
2     for i in range(n):
3         print('pif')
4     for j in range(n):
5         print('paf')
6     for k in range(n):
7         print('pouf')
```

```
1 def fonction2(n):
2     for i in range(n):
3         print('pif')
4         for j in range(i):
5             print('paf')
6     for k in range(n):
7         print('pouf')
```

```
1 def fonction3(n):
2     for i in range(n):
3         print('pif')
4         for j in range(n):
5             print('paf')
6             for k in range(n):
7                 print('pouf')
```

Pour ces trois fonctions, on mesure la complexité par le nombre d'affichages réalisés en fonction de n :

Appel de	Nombre d'affichages réalisés
`fonction1(n)`	$3n$
`fonction2(n).`	$n + \dfrac{n(n+1)}{2}$
`fonction3(n)`	$n\left(1 + n\left(n+1\right)\right) = n^3 + n^2 + n$

Détaillons le calcul par exemple pour `fonction2(n)`. Le corps de la boucle `for i` réalise exactement `i` $+ 1$ affichages. Sachant que `i` prend les valeurs successives $0, 1, ..., n-1$, la boucle `for i` va réaliser en tout

$$\sum_{i=0}^{n-1} (i+1) = \frac{n(n+1)}{2}$$

affichages. La boucle `for k`, qui est réalisée une fois la boucle `for i` terminée, produit quant à elle n affichages.

2 Evaluation asymptotique de la complexité

Dans les exemples précédents, on a déterminé de façon exacte le nombre des opérations effectuées par chacun des algorithmes. En réalité, il est souvent inutile d'aller jusqu'à ce niveau de détail pour évaluer l'efficacité d'un algorithme.

Déjà, le résultat étant significatif seulement pour les grandes valeurs de n, il suffit d'obtenir une évaluation asymptotique du nombre d'opérations effectuées.

De plus, les différentes opérations considérées ne demandent pas toutes le même temps de calcul et cachent donc un facteur multiplicatif difficile à déterminer. C'est pourquoi on se contente souvent de majorer le nombre d'opérations par le produit d'une constante et d'une fonction élémentaire de n.

Conformément au programme de mathématiques, on dira qu'un algorithme a une complexité en $O(f(n))$ si son coût est, à partir d'un certain rang, inférieur au produit de $f(n)$ par une constante.

Le tableau suivant fait l'inventaire des complexités que nous allons rencontrer dans la suite.

	Nom courant
$O(1)$	Temps constant
$O(\ln(n))$	Complexité logarithmique
$O(n)$	Complexité linéaire
$O(n^2)$	Complexité quadratique

Exercice 7

Dans les algorithmes déjà rencontrés dans ce cours, préciser si la complexité est linéaire, quadratique...

Exercice 8

Ecrire une fonction qui prend comme argument une liste de flottants et qui renvoie son plus grand élément. Quel est sa complexité ?

```
1 def maximum(L):
2     M=L[0]
3     for x in L:
4         if x>M:
5             M=x
6     return M
```

On mesure la complexité en comptant le nombre de comparaisons de flottants en fonction de la taille n de la liste. La liste étant parcourue une seule fois, la complexité est linéaire.

Exercice 9

Une matrice carrée de taille n peut être représentée par une liste de n listes de flottants.

Par exemple, la matrice $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$ est représentée par `L=[[1,2,3],[4,5,6],[7,8,9]]`.

Ecrire une fonction `Moyenne` qui prend en argument une matrice carrée de taille n et qui renvoie la moyenne de ses coefficients. Quelle est sa complexité ?

```
1 def Moyenne(L):
2     S=0
3     n=len(L)*
4     for l in L:
5         for x in l:
6             S+=x
7     return S/n**2
```

La complexité est évaluée par le nombre d'opérations algébriques en fonction de n. La matrice étant parcourue une seule fois et comportant n^2 coefficients, la complexité est quadratique.

3 Complexité et récursivité

On considère la suite $(u_n)_{n\in\mathbb{N}}$ définie par récurrence par :

$$u_0 = 2 \text{ et } \forall n \geq 1, u_n = \frac{u_{n-1}}{2} + \frac{1}{u_{n-1}}.$$

On considère les deux fonctions récursives :

```
1 def u(n):
2     if n==0:
3         return 2
4     else:
5         A=u(n-1)
6         return A/2+1/A
```

```
1 def u1(n):
2     if n==0:
3         return 2
4     else:
5         return u1(n-1)/2+1/u1(n-1)
```

Évaluer la complexité de chaque fonction en comptant le nombre d'opérations algébriques effectuées lors de l'appel de $u(n)$ et $u1(n)$. Commenter.

Exercice 10

On considère la suite de Fibonacci :

$$f_0 = 1 \qquad f_1 = 1 \qquad \forall p \in \mathbb{N}, f_{p+2} = f_{p+1} + f_p.$$

1. Écrire une fonction non récursive `fibo(n)` renvoyant le terme f_n de rang n de la suite de Fibonacci. Évaluer sa complexité temporelle.
2. Écrire une fonction récursive `fiboRec` f_n. Évaluer la complexité temporelle.
3. Commenter.

4 Complexité et principe dichotomique

4.A Écriture binaire d'un nombre entier

La fonction suivante renvoie l'écriture binaire sous forme d'une liste d'un entier $n \geq 1$ en argument. Ainsi, `binaire(14)` renvoie la liste `[1,1,1,0]` avec $14 = 2^3 + 2^2 + 2^1 + 0 \times 2^0$.

Evaluer sa complexité.

```
1 def binaire(n) :
2     nombre=n
3     liste=[]
4     while nombre>0 :
5         liste=[nombre%2] +liste
6         nombre=nombre//2
7     return(liste)
```

Pour tout entier $n \in \mathbb{N}^*$, il existe un unique $k \in \mathbb{N}$, tel que $2^k \leq n < 2^{k+1}$.

Dans le programme `binaire`, il y a un test, une division par deux, un modulo et une concaténation par passage dans la boucle while. Donc pour évaluer la complexité, il faut connaître le nombre de passages.

A chaque passage, la variable `nombre` est divisée par 2, jusqu'à ce que nombre soit égal à 0.

Donc avec $2^k \leq n < 2^{k+1}$, le nombre d'étapes sera égal à $k+1$.

Pour exprimer k en fonction de n, on remarque $k\ln(2) \leq \ln(n) < (k+1)\ln(2)$ donc, avec $\ln(2) > 0$, il vient

$$k \leq \frac{\ln(n)}{\ln(2)} < k+1 \text{ et ainsi } k = \left\lfloor \frac{\ln(n)}{\ln(2)} \right\rfloor \text{ la partie entière de } \frac{\ln(n)}{\ln(2)}.$$

Bilan : la complexité de `binaire` est logarithmique en n.

4.B Recherche dans une liste triée

```
1 def rechercheDicho(x,a):
2     g,d=0,len(a)-1
3     while d>=g:
4         m=(d+g)//2
```

```
5           if a[m]==x:
6               return True
7           elif x<a[m]:
8               d=m-1
9           else:
10              g=m+1
11      return False
```

Lors de chaque itération de la boucle `while`, au plus 2 comparaisons sont effectuées.
On montre sans difficulté par récurrence que, à la fin de la k-ième itération de la boucle, $d-g \leq \frac{n}{2^k}$.
Si $\frac{n}{2^k} < 1$, i.e. si $k > \frac{\ln(n)}{\ln(2)}$, alors $d-g \leq 0$: il reste donc au plus une itération.
Par conséquent, la complexité de `rechercheDicho` est en $O(\ln(n))$.

5 Différentes notions de complexité

5.A Complexité dans le pire et dans le meilleur des cas

Pour deux données de même taille, un algorithme n'effectue pas nécessairement le même nombre d'opérations élémentaires. Par exemple, considérons le test d'appartenance à une liste vu au chapitre précédent :

```
1 def appartient(x,L):#teste si x appartient à la liste L
2     for y in L:
3         if y==x:
4             return True
5     return False
```

L'appel `appartient(x,L)` réalise au mieux une seule comparaison (cas où x est présent en première position dans `L`), et au pire $n =$ `len(L)` comparaisons (cas où x est absent ou en dernière position dans `L`).

Cet exemple illustre la notion de complexité dans le meilleur cas qui est le nombre d'opérations minimal pour effectuer le programme. La complexité dans le pire cas est le nombre d'opérations maximal.

5.B Complexité en espace

Jusqu'ici, on a uniquement discuté du temps d'exécution des algorithmes. Une autre ressource importante en informatique est la mémoire. On appelle **complexité en espace** d'un algorithme la place nécessaire en mémoire pour le faire fonctionner. Elle s'exprime également sous la forme d'un $O(f(n))$ où n est la taille du problème.

Exercice 11

Le programme suivant retourne les n premières lignes du tableau de Newton qui contient les coefficients binomiaux. La liste en retour est formée de listes, celle d'indice k contient les $\binom{k}{j}$ pour $0 \leq j \leq k$. Par exemple, en notant l la liste résultat, $l[3]$ vaut $[1, 3, 3, 1]$ et $l[3][0] = \binom{3}{0}$, $l[3][1] = \binom{3}{1}$, etc. Pour le code de cette fonction, on a utilisé la relation de Pascal :

$$\forall k \geq 2, \forall j \in [\![1, k-1]\!], \binom{k}{j} = \binom{k-1}{j-1} + \binom{k-1}{j}.$$

```python
def tableauNewton(n) :
    liste=[[1]] # "0 parmi 0" vaut 1
    for k in range(1,n) :
        aux=[1] # "0 parmi k" vaut 1
        for j in range(1,k) :
            aux.append(liste[k-1][j-1]+liste[k-1][j])
            #calcul de "j parmi k" avec la
            # formule récurrente pour j entre 1 et k-1
        aux.append(1) # "k parmi k" vaut 1
        liste.append(aux)
        # on ajoute cette ligne du tableau à la liste résultat
    return(liste)
```

```
>>> tableauNewton(5)
[[1], [1, 1], [1, 2, 1], [1, 3, 3, 1], [1, 4, 6, 4, 1]]
```

Cette fonction crée une liste de listes, la liste d'indice k étant de longueur $k+1$. Donc, le nombre d'entiers stockés en mémoire est de l'ordre de :

$$\sum k = 0^{n-1}(k+1) = \frac{n(n+1)}{2}.$$