Analyse et méthodes numériques, module M2202

Pierre-Cyrille Héam

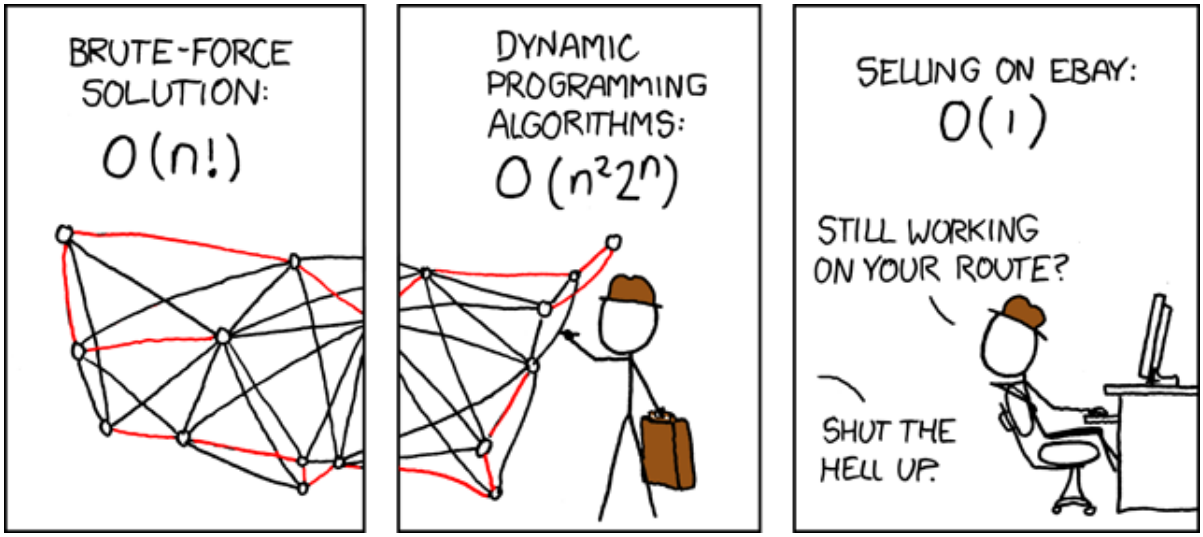
18 décembre 2014

Chapitre 1

Introduction

L'objectif du module (module M2202 du PPN) est *comprendre les notions fondamentales de l'approximation et de la convergence.* Les compétences visées sont *Majorer, minorer, gérer les approximations.* Le contenu *Suites et fonctions numériques, limites et convergence, comportement local.*

Le choix est fait ici d'aborder ce module pour une de ses applications fondamentale en informatique : la complexité algorithmique, qui demande des calculs sur des suites, des comparaisons de fonctions, des approximations, et qui s'appuie sur les notations de Landau (un des développement proposé dans le PPN). A la fin du module, il devrait être possible de comprendre l'humour *geek* suivant [1] :



1.1 Efficacité des algorithmes

D'après vous qu'est-ce qu'un algorithme efficace ?, comment mesurer cette efficacité ?

On peut distinguer plusieurs façons de mesurer qu'un algorithme est efficace :

— La facilité/rapidité d'implémentation. C'est bien sûr un critère, à prendre en compte lorsque l'on doit mettre en oeuvre un programme. Cependant, il s'agit d'une notion assez difficile à mesurer précisément, puisque cela dépend du langage de programmation et des bibliothèques disponibles. Il y a des théories de ce type de questions, appelées *complexité de Kolmogorov* ou *complexité descriptive*. Il s'agit de travaux très théoriques, difficiles et dont les applications sont difficiles à appréhender.

— L'utilisation de ressources : un bon algorithme est un algorithme qui s'exécute rapidement (il consomme peu de temps) et qui utilise peu d'espace mémoire. Pour certaines applications, sur un système mobile à batteries, la consommation d'énergie peut aussi être prise en compte : on peut accepter un algorithme un peu plus lent s'il consomme moins. Nous nous intéresserons dans ce cours à ce type de complexité, en regardant essentiellement la complexité en temps de calcul (l'étude des autre ressources pourrait se faire de façon similaire).

1. `http://xkcd.com/399/`

Exercice 1 *On considère l'algorithme suivant :*

```
def fonction1 (x,n):
   res = x
   for i in range(1,n):
      res=res*x
   return res
```

1. *Que retourne cet algorithme ? (en fonction de x et n)*
2. *Même question avec l'algorithme suivant (ou pourra par exemple essayer avec $n = 6$) :*

```
def fonction2 (x,n):
   if n == 0:
       return 1
   if n est pair:
       res=fonction2(x,n/2)
       return res*res
   else :
      res=fonction2(x,(n-1)/2)
      return res*res*x
```

3. *Même question avec l'algorithme suivant :*

```
def fonction3 (x,n):
   if n == 1:
       return x
   if n est pair:
       return fonction3(x,n/2)*fonction3(x,n/2)
   else :
       return fonction3(x,(n-1)/2)*fonction3(x,n/2)*x
```

4. *Classer, intuitivement, ces algorithme du plus efficace au moins efficace (vous pouvez vous tromper, il ne s'agit que de mesurer une intuition).*
5. *Le plus efficace est-il un peu plus efficace que le second ou beaucoup plus efficace ?*
6. *Le second est-il un peu plus efficace que le troisième ou beaucoup plus efficace ?*

1.2 Complexité en temps

Le temps qu'un programme met pour s'exécuter dépend de la machine utilisée, du langage, de l'environnement, etc. On ne cherche pas ici à mesurer le temps d'exécution d'un programme mais d'un algorithme. Pour cela on suppose qu'il y a un certains nombre d'opérations élémentaires qui prennent une unité de temps et l'on compte ce nombre d'opérations élémentaires. En général, les opérations élémentaires sont :

— Une addition, une multiplication, une division, etc sur des variables numériques,
— La comparaison de deux variables,
— L'affectation d'une variable.
— La lecture d'une variable.
— ...

Dans certains cas, on n'utilise que certaines opérations élémentaires, mais cela est alors précisé. Il y a là une approximation, car le coût d'une addition n'est pas le même que celui d'une multiplication, mais on s'intéresse ici à des ordres de grandeur.

Considérons par exemple la fonction `fonction1` de l'exercice 1 le nombre d'opérations élémentaires est :

— 1 pour `res=x`
— à chaque itération de la boucle, la valeur de `i` est incrémentée (une opération) et affectée à `i` (une opération), plus une multiplication et une affection chaque fois que `res=res*x` est exécutée. Donc 4 opérations élémentaires à chaque itération.
— 1 pour le `return`

Au total cela fait donc $1 + 4(n-1) + 1$ car on effectue $n-1$ fois la boucle. Cet algorithme consomme donc $4n-2$ opérations élémentaires (si $n \geq 1$).

Exercice 2 *Donner en fonction de n, le nombre d'opérations élémentaires utilisées dans l'algorithme ci-dessous.*

```
def fonction4(n):
   res=0
   for i in range(n):
      res=res+i
   return res
```

Exercice 3 *Donner en fonction de n, le nombre d'opérations élémentaires utilisées dans l'algorithme ci-dessous.*

```
def fonction5(n):
   res=0
   for i in range(n):
      for j in range(n):
         res=res+i*j
   return res
```

Exercice 4 *Donner en fonction de n, le nombre d'opérations élémentaires utilisées dans l'algorithme ci-dessous.*

```
def fonction6(n):
   res=1
   k=n*n
   for i in range(k):
      res=res*i
   return res
```

Exercice 5 *Donner en fonction de n, le nombre d'opérations élémentaires utilisées dans l'algorithme ci-dessous.*

```
def fonction7(n):
   res=1
   for i in range(n*n):
      res=res*i
   return res
```

On s'intéresse maintenant à l'algorithme `fonction2` de l'exercice 1 et on note $T(n)$ le nombre d'opérations pour une entrée n (la valeur de x n'est pas importante ici vu le mode de calcul). On a alors

— $T(1) = 2$,
— Si n est pair $T(n) = 5 + T(n/2)$,
— Si n est impair $T(n) = 6 + T((n-1)/2)$.

Avec un raisonnement approximatif on a

$$T(n) \simeq 6 + T(n/2) \simeq 6 + 6 + T(n/4) \simeq 6 + 6 + 6 + T(n/8) \simeq \ldots \simeq 6 + 6 + 6 + \ldots + 6 + T(1) \simeq 6\log_2(n).$$

Le nombre d'opérations utilisés pour la fonction2 est donc de l'ordre de $6\log_2(n)$.

Si l'on s'intéresse maintenant à la fonction `fonction3`, en notant $T'(n)$ le nombre d'opérations, on a

— $T'(1) = 2$,
— Si n est pair $T'(n) = 5 + 2 * T'(n/2)$,
— Si n est impair $T'(n) = 6 + 2 * T'((n-1)/2)$.

Avec un raisonnement approximatif on a

$$T'(n) \simeq 6 + 2 * T'(n/2) \simeq 6 + 2 * (6 + T'(n/4)) \simeq 6 + 2 * 6 + 2 * (6 + T'(n/8)) \simeq \ldots \simeq 6 + 6 + 6 + \ldots + 6 + T'(1) \simeq 6n.$$

Exercice 6

1. *En utilisant votre calculatrice, remplir le tableau suivant en inscrivant la valeur approximative des fonctions données pour les valeurs de n données :*

n	*1*	*8*	*16*	*32*	*1024*	2^{20}	2^{1000}	2^{2048}
$4n-2$								
$6n$								
$6\log_2(n)$								

2. *On suppose maintenant que l'on dispose d'un ordinateur qui calcule 12 milliards d'opérations par secondes (12 GHz). Remplir le tableau suivant du temps qu'il mettrait pour calculer les fonctions décrites pour les valeurs de n données. Donner les temps dans l'unité de temps qui vous semble à chaque fois la plus appropriée pour être lisible.*

n	*1*	*8*	*16*	*32*	*1024*	2^{20}	2^{1000}	2^{2048}
`fonction1`								
`fonction2`								
`fonction3`								

3. *Qu'en déduisez vous ?*

Il faut noter plusieurs choses :
— *Un ordinateur a besoin de plusieurs cycles pour calculer le produit de deux entiers.*

— *Ici les entiers deviennent vite très long, rendant en fait impossibles les calculs. En pratique cela est cependant utilise, sauf qu'à chaque étape le résultat est tronqué (à l'aide d'un modulo). Les ordres de grandeurs ont donc un sens.*
— *Prendre une entrée* $n = 2^{1000}$ *peut paraître étrange, mais ce type de puissance (avec des modulo) est effectivement calculé en cryptographie. Par exemple lorsque vous faite un* `ssh` *ou un* `https`*, des calculs de* x^n *(avec modulo) sont effectuées, où* n *est la clé de chiffrement. Si vous avez une clé sur 1024 bits, on a* n *qui est de l'ordre de* 2^{1024}*.*

Exercice 7 *En 1 minute, votre programme peut résoudre un problème de taille maximum* M*. La compagnie* X *vous propose d'acheter son nouveau microprocesseur, 100 fois plus rapide que le précédent. Si vous achetez le nouveau microprocesseur, en 1 minute votre programme pourra résoudre des problèmes de taille plus grande. Donnez une fourchette de la nouvelle taille maximale si la complexité (nombre d'opérations élémentaires) de votre algorithme est :* n*,* $2n$*,* $3n$*,* n^2*,* n^4*,* 2^n *?*

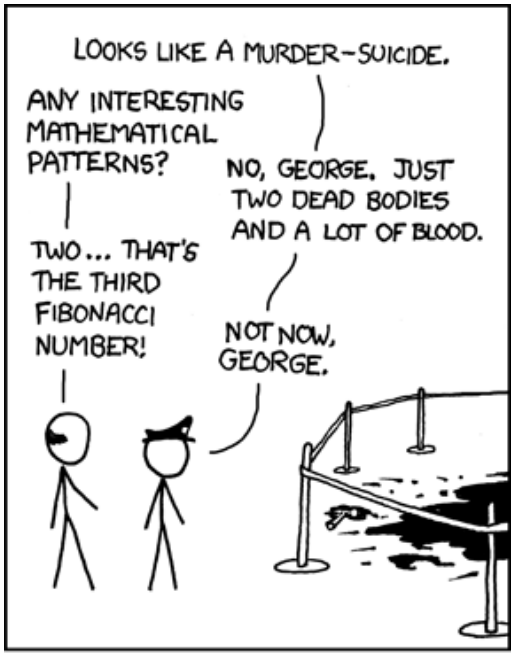
1.3 Conclusion

Comme on a pu le voir, de petits calculs montrent des différences de comportement très importantes entre divers algorithmes (pourtant très simples) pour calculer x^n. Si pour de petites valeurs de n, il y a peu d'enjeu (sauf si la fonction est appelées beaucoup de fois), il en va autrement pour les grandes valeurs.

Nous allons voir dans ce cours
— Que le calcul des complexités (nombre d'opérations) passe souvent par une étude de suites récurrentes.
— Que les complexités sont souvent difficiles à calculer explicitement et, comme on s'intéresse à des ordres de grandeurs, qu'on les comptes approximativement. Nous verrons donc avec quelles fonctions l'on fait cela et comment calculer et manipuler ces approximations.

Chapitre 2

Suites récurrentes



WHEN MATHNET SHUT DOWN, THE OFFICERS HAD TROUBLE REINTEGRATING INTO THE REGULAR L.A.P.D. [1]

2.1 Motivations

Exercice 8 *Mon banquier me propose un placement étrange. Je place un euro. Et chaque mois mon placement gagne 4 fois la somme qu'il contenait le mois d'avant moins 6 fois celle qu'il contenait 2 mois avant. Si une somme devient négative, ça n'est pas un problème, je dois juste de l'argent à mon baquier. Comme la procédure ne marche pas le second mois, le banquier m'accorde un euro supplémentaire le second mois. Je me demande quelle somme j'aurais au bout de n mois.*

1. *On considère donc la suite définie par*

$$u_{n+2} = 5u_{n+1} - 6u_n \quad et \quad u_0 = 1 \quad u_1 = 2.$$

   Justifier pourquoi elle répond au problème.
2. *Calculer u_i pour i de 0 à 6.*
3. *On souhaite calculer u_n, on propose pour ça trois fonctions. A votre avis, laquelle auriez-vous spontanément programmée (en supposant que vous n'ayez pas fait la question 1) ? Calculer les premiers résultats de chaque fonctions pour les premiers termes pour vous convaincre que les résultats semblent bien les mêmes.*

```
def fu1 (n):
    if n == 0 :
        return 1
    if n == 1 :
        return 2
    return 5*fu1(n-1)-6*fu1(n-2)
```

1. http://xkcd.com/587/

```
def fu2(n):
    if n == 0 :
        return 1
    if n == 1 :
        return 2
    res=[1,2]
    i = 2
    while i <= n:
        res.append(5*res[i-1]-6*res[i-2])
        i=i+1
    return res[n]


def fu3(n):
    return 2**n
```

4. *On note $T_1(n)$, $T_2(n)$, $T_3(n)$ le nombre d'opérations utilisées par les fonctions* `fu1`, `fu2` *et* `fu3` *Exprimer $T_1(n+2)$ en fonction $T_1(n+1)$ et $T_1(n)$. En calculant $T_1(n+1) - T_1(n)$ justifier que T_1 est une fonction croissante. Justifier qu'il existe une constante c_1 telle que $T_1(n) \geq c_1 2^{n/2}$.*
5. *Justifier qu'il existe deux constantes c_2 et c_3 telles que*

$$c_2 * n \leq T_2(n) \leq c_3 * n$$

6. *En vous reportant au chapitre précèdent que vaut approximativement $T_3(n)$?*
7. *Remplir le tableau suivant en vous aidant de votre calculatrice.*

n	*1*	*8*	*16*	*32*	*1024*	2^{20}	2^{1000}	2^{2048}
$2^{n/2}$								
$7n$								
$6\log_2(n)$								

8. *On suppose maintenant que l'on dispose d'un ordinateur qui calcule 12 milliards d'opérations par secondes (12 GHz). Remplir le tableau suivant du temps approximatif qu'il mettrait pour calculer les 3 fonctions.*

n	*1*	*8*	*16*	*32*	*1024*	2^{20}	2^{1000}	2^{2048}
`fu1`								
`fu2`								
`fu3`								

9. *Qu'en déduisez vous ?*
10. *On a donné une minoration de $T_1(n)$. Sauriez vous calculer plus précisément $T_1(n)$?*

2.2 Rappel sur les suites

Definition 9 *Une* **suite** u_n *à valeurs réelles est une application de $\mathbb{N}$ dans $\mathbb{R}$.*

On note u_n l'image de n par u ; plutôt que (u_n). D'ailleurs la suite u est souvent notée $(u_n)_{n\in\mathbb{N}}$ ou simplement (u_n).

Par exemple la suite u qui à n associe $2n+1$ vérifie $u_0 = 1$, $u_1 = 3$, $u_{17} = 35$, etc.

Dans ce document, toutes les suites sont à valeur dans $\mathbb{R}$, on parlera donc juste de *suite* à la place de *suite à valeurs réelles.*

Comme sur toute application à valeurs réelles on peut faire la somme, la différence, et le produit de deux suites.

Exercice 10 *Donner les cinq premiers termes des suites suivantes :*

1. $u_n = \frac{n+1}{n+2}$,
2. $u_n = 1 + n^2$,
3. $u_0 = 1$ *et* $u_{n+1} = u_n^2 - 1$.

Exercice 11 *On considère la suite (u_n) définie par $u_n = 3n + 2^n$ et la suite (v_n) définie par $v_n = -2n + \frac{1}{n^2+1}$. Que valent les suites $(u_n + v_n)$ et $(u_n v_n)$?*

Exercice 12 *On considère la suite définie par $u_n = 2n + 3$. Donner, en fonction de n, les valeurs de u_{n+1}, $u_n + 1$, u_{2n}, u_{2n+1}, u_{n^2}, u_n^2 et $2u_n$.*

Definition 13 *Toute suite (u_n) vérifiant $u_{n+1} = u_n + r$ où r est une constante indépendant de n est appelée* **suite arithmétique**. *Le réel r est appelée* **raison** *de la suite.*

Par exemple la suite (u_n) définie par $u_0 = 0$ et $u_{n+1} = u_n + 3$ est la suite arithmétique de raison 3 et de premier terme 0.

Exercice 14

1. *Que vaut le cinquième terme de la suite arithmétique de raison 2 et premier terme 3.*
2. *On considère une suite arithmétique de raison r et premier terme a. Exprimer le n-ième terme en fonction de r et de a.*
3. *On considère la suite arithmétique (u_n) de premier terme 0 et raison 1. Que vaut u_n ?*
4. *On pose*
$$S = u_0 + u_1 + \ldots + u_n = \sum_{i=0}^{n} u_i.$$
En couplant les éléments deux à deux, donner la valeur de $2S$. En déduire la valeur de S.
5. *On considère une suite (v_n) arithmétique de raison r. On pose*
$$T_n = v_0 + v_1 + \ldots + v_n = \sum_{i=0}^{n} v_i.$$
En utilisant la même méthode que ci-dessus, exprimer $2T_n$ en fonction de u_0, u_n et n. En déduire la valeur de T_n en fonction de n et u_0.

Proposition 15 *Soit (v_n) une suite arithmétique de raison r. On a*

$$v_n =$$

et

$$\sum_{i=0}^{n} v_i =$$

Exercice 16

1. *On considère la suite définie par $u_{n+1} = u_n + 2$ et $u_0 = 1$, que vaut u_n en fonction de n ? Que vaut la somme des $n+1$ premiers termes de la suite (u_n) ?*
2. *Mêmes questions avec $v_{n+1} = v_n - 3$ et $v_0 = -1$.*

Exercice 17 *On considère la fonction suivante :*

```
def toto(n):
   i=1
   while i <= n:
      j=1:
      while j <= i:
         print "hello"
         print "world"
         i=i+1
         j=j+1
```

1. *On note T_n le nombre de fois que la fonction* `print` *est appelée par la fonction* `toto`. *Que vaut $T(1)$?*
2. *Combien de fois* `toto(n+1)` *appelle de fois la fonction* `print` *de plus que* `toto(n)` *?*
3. *En déduire la valeur de $T(n)$.*

Definition 18 *Toute suite (u_n) vérifiant $u_{n+1} = qu_n$ où q est une constante indépendant de n est appelée* **suite géométrique**. *Le réel q est appelée* **raison** *de la suite.*

Par exemple la suite définie par $u_{n+1} = 5u_n$ et $u_0 = 2$ est une suite géométrique de raison 5.

Exercice 19

1. *Que vaut le cinquième terme de la suite géométrique de raison 2 et premier terme $u_0 = 1$.*
2. *On considère une suite géométrique de raison q et premier terme a. Exprimer le n-ième terme en fonction de q et de a.*
3. *En développant, simplifier $(1-q)(1+q+q^2+\ldots+q^n)$.*
4. *En déduire une expression de $1+q+q^2+\ldots+q^n$ lorsque $q \neq 1$.*
5. *Soit (u_n) une suite géométrique de raison 1. Que vaut u_n en fonction de u_0. Que vaut dans de cas $u_0 + u_1 + \ldots + u_n$?*

6. *Soit (u_n) une suite géométrique de raison $q \neq 1$. Que vaut dans de cas $u_0 + u_1 + \ldots + u_n$?*

Proposition 20 *Soit (v_n) une suite géométrique de raison $q \neq 1$. On a*

$$v_n =$$

et

$$\sum_{i=0}^{n} v_i =$$

Exercice 21
1. *On considère la suite définie par $u_{n+1} = 2u_n$ et $u_0 = 3$, que vaut u_n en fonction de n ? Que vaut la somme des $n+1$ premiers termes de la suite (u_n) ?*
2. *Mêmes questions avec $v_{n+1} = -2u_n$ et $v_0 = 1$.*

Exercice 22 *On considère la fonction suivante :*

```
def titi(n):
   if n == 1 :
      return "a"
   m=""
   m=m+titi(n-1)+titi(n-1)+titi(n-1)
```

1. *On note ℓ_n la longueur de la chaîne retournée par* `titi(n)`*. Que vaut ℓ_1 ? ℓ_2 ? ℓ_3 ?*
2. *Exprimer ℓ_n en fonction de ℓ_{n-1}.*
3. *En déduire la valeur de ℓ_n.*
4. *On considère la fonction suivante :*

```
def tata(n):
   if n == 1 :
      return "a"
   m=""
   i=1
   while i <= n:
      m=m+titi(i)
      i=i+1
   return m
```

Donner, en fonction de n la taille de la chaîne retournée par `tata`*.*

Definition 23 *Une suite (u_n) est* **croissante** *si quelque soit n, $u_{n+1} - u_n \geq 0$. Elle est* **strictement croissante** *si quelque soit n, $u_{n+1} - u_n > 0$. Elle est* **décroissante** *si quelque soit n, $u_{n+1} - u_n \leq 0$ et* **strictement croissante** *si quelque soit n, $u_{n+1} - u_n < 0$.*

2.3 Suite arithmético-géométrique

Definition 24 *Une* **suite arithmético-géométrique** *est une suite (u_n) vérifiant $u_{n+1} = au_n + b$ où a et b sont deux constantes réelles (indépendantes de n).*

On supposera que $a \neq 0$, car sinon la suite est constante.

Par exemple la suite définie par $u_{n+1} = 3u_n - 2$ est une suite récurrente d'ordre 1.

Si $a = 1$ dans la définition précédente, alors la suite (u_n) est une suite arithmétique et si $b = 0$, alors c'est une suite géométrique.

On va voir que l'étude des suites récurrentes d'ordre 1 se ramène en général à l'étude des suites géométriques. Soit (u_n) vérifiant $u_{n+1} = au_n + b$ avec $a \neq 1$:

— On pose $\alpha = \frac{b}{1-a}$, ce qui est possible puisque $a \neq 0$.

— Calculons maintenant $u_{n+1} - \alpha$. On a

$$\begin{aligned} u_{n+1} - \alpha &= au_n + b - \alpha = au_n + b - \alpha + a\alpha - a\alpha \\ &= au_n - a\alpha + b - \alpha + a\alpha = a(u_n - \alpha) + b - \alpha + a\alpha \\ &= a(u_n - \alpha) + b + (a-1)\alpha = a(u_n - \alpha) + \frac{(1-a)b}{1-a} + \frac{(a-1)b}{1-a} \\ &= a(u_n - \alpha) \end{aligned}$$

— en posant $v_n = u_n - \alpha$, la suite (v_n) est donc une suite géométrique de raison a. On a donc

$$v_n = v_0 a^n = (u_0 - \alpha)a^n = (u_0 - \frac{b}{1-a})a^n,$$

donc

$$u_n = (u_0 - \frac{b}{1-a})a^n + \frac{b}{1-a}.$$

— Par ailleurs, $u_0 + u_1 + \ldots + u_n = v_0 + \alpha + v_1 + \alpha + \ldots v_n + \alpha$ et $v_0 + v_1 + \ldots + v_n = v_0 \frac{1-a^{n+1}}{1-a}$. Donc

$$u_0 + u_1 + \ldots + u_n = (u_0 - \frac{b}{1-a})\frac{1-a^{n+1}}{1-a} + (n+1)\frac{b}{1-a}.$$

Proposition 25 *Soit (u_n) une suite vérifiant $u_{n+1} = au_n + b$ avec $a \neq 1$. On a*

$$u_n = (u_0 - \frac{b}{1-a})a^n + \frac{b}{1-a},$$

et

$$u_0 + u_1 + \ldots + u_n = (u_0 - \frac{b}{1-a})\frac{1-a^{n+1}}{1-a} + (n+1)\frac{b}{1-a}.$$

Exercice 26

1. *On considère la suite définie par $u_{n+1} = 3u_n + 2$ et $u_0 = 0$; que vaut u_n en fonction de n ? Que vaut la somme des $n+1$ premiers termes de la suite (u_n) ?*
2. *Mêmes questions avec $v_{n+1} = -u_n + 3$ et $v_0 = 1$.*
3. *Mêmes questions avec $x_{n+1} = 4u_n - 2$ et $x_0 = -2$.*

Exercice 27 *On considère les fonctions suivantes :*

```
def tonton(n):
    if n > 0:
        print "Hello world"
        tonton(n-1)
        tonton(n-1)

def tutu(n):
    if n > 0:
        for i in range(n+1):
            tonton(i)
```

1. *Qu'affiche* `tonton(2)` *?*
2. *Qu'affiche* `tonton(3)` *?*
3. *On note u_n le nombre d'appels à la fonction* `print` *de* `tonton(n)`*. Montrer que (u_n) est une suite récurrente d'ordre 1 dont on précisera les paramètres.*
4. *Que vaut u_n en fonction de n ?*
5. *Donner en fonction de n le nombre d'appels à la fonction* `print` *de* `tutu(n)`*.*

Exercice 28 [2] *Le nombre d'arbres d'une forêt, en milliers d'unités, est modélisé par la suite (u_n) où u_n désigne le nombre d'arbres, en milliers, au cours de l'année (2010 + n). En 2010, la forêt possède 50 000 arbres. Afin d'entretenir cette forêt vieillissante, un organisme régional d'entretien des forêts décide d'abattre chaque année 5 % des arbres existants et de replanter 3 000 arbres.*

1. *Que vaut u_0, u_1 ?*
2. *Exprimer u_{n+1} en fonction de u_n.*
3. *En déduire la valeur de u_n en fonction de n.*
4. *Déterminer l'année à partir de laquelle le nombre d'arbres de la forêt aura dépassé de 10% le nombre d'arbres de la forêt en 2010.*
5. *À long terme, la forêt se stabilisera à combien d'arbres ?*

2. Exercice inspiré de `http://www.jybaudot.fr/Suites/exarithmetgeo.html`

2.4 Suites récurrentes d'ordre 2

Definition 29 *Une* **suite récurrente d'ordre 2** *est une suite (u_n) vérifiant $u_{n+2} = \alpha u_{n+1} + \beta u_n$ où α et β sont deux constantes réelles (indépendantes de n).*

Par exemple la suite définie par $y_{n+2} = 4y_{n+1} - 4y_n$ et $y_0 = 1$ et $y_1 = 3$ est une suite récurrente d'ordre 2. De même la suite $x_{n+2} = \frac{5}{2}x_{n+1} - x_n$ et $x_0 = x_1 = 1$ est un autre exemple.

Pour exprimer la valeur d'une suite récurrente d'ordre 2 définie en fonction de n, la méthode générale consiste à :

1. Considérer l'équation $x^2 - \alpha x - \beta = 0$. Pour la suite (y_n) on a $\alpha = 2$ et $\beta = -1$, c'est donc l'équation $x^2 - 4x + 4 = 0$. Pour la suite (x_n) c'est l'équation $x^2 - \frac{5}{2} + 1 = 0$. On calcule le discriminant Δ de cette équation. Compte tenu de la forme de l'équation, on a $\Delta = \alpha^2 + 4\beta$.
2. **Si Δ est strictement positif**, on sait que l'équation a deux racines réelles distinctes r_1 et r_2, qui valent
$$r_1 = \frac{\alpha + \sqrt{\alpha^2 + 4\beta}}{2} \quad \text{et} \quad r_1 = \frac{\alpha - \sqrt{\alpha^2 + 4\beta}}{2}.$$
D'après la théorie mathématique, on sait alors que la suite vérifie que son n-ième terme vaut $ar_1^n + br_2^n$, où a et b sont deux constantes (indépendantes de n) que l'on peut définir grâce à u_0 et u_1 (par exemple) ; en prenant $n = 0$ on trouve que $u_0 = a + b$ et en prenant $n = 1$ on trouve que $u_1 = ar_1 + br_2$. On trouve donc a et b en résolvant le système :
$$\left\{ \begin{array}{rcl} a + b & = & u_0 \\ r_1 a + r_2 b & = & u_1 \end{array} \right.$$
Par exemple, dans le cas de la suite (x_n), on a
$$\Delta = \left(\frac{5}{2}\right)^2 + 4(-1) = \frac{25}{4} - 4 = \frac{25}{4} - \frac{16}{4} = \frac{9}{4} = \left(\frac{3}{2}\right)^2,$$
donc
$$r_1 = \frac{\frac{5}{2} + \sqrt{\left(\frac{3}{2}\right)^2}}{2} = \frac{\frac{5}{2} + \frac{3}{2}}{2} = \frac{\frac{8}{2}}{2} = \frac{4}{2} = 2,$$
et
$$r_2 = \frac{\frac{5}{2} - \sqrt{\left(\frac{3}{2}\right)^2}}{2} = \frac{\frac{5}{2} - \frac{3}{2}}{2} = \frac{\frac{2}{2}}{2} = \frac{1}{2}.$$
Dans ce cas, on sait qu'il existe a et b tels que pour tout n, $x_n = ar_1^n + br_2^n = a2^n + b\frac{1}{2^n}$. On résout le système
$$\left\{ \begin{array}{rcl} a + b & = & x_0 \\ r_1 a + r_2 b & = & x_1 \end{array} \right. = \left\{ \begin{array}{rcl} a + b & = & 1 \\ 2a + \frac{1}{2}b & = & 1 \end{array} \right.$$
On trouve alors que $a = \frac{1}{3}$ et $b = \frac{2}{3}$. On a donc
$$x_n = \frac{1}{3}2^n + \frac{2}{3}\left(\frac{1}{2}\right)^n$$
3. **Si Δ est égal à zéro**, l'équation a une unique racine (double) qui vaut $r = \frac{\alpha}{2}$. D'après la théorie mathématique, on sait alors que la suite vérifie que le n-ième terme vaut $(an+b)r^n$, où a et b sont deux constantes (indépendantes de n) que l'on peut définir grâce à u_0 et u_1 ; en prenant $n = 0$ on trouve que
$$u_0 = b$$
et en prenant $n = 1$ on trouve que $u_1 = (a + b)r = (a + u_0)r$. Comme on a supposé $\alpha \neq 0$, on a $r \neq 0$ et donc
$$a = \frac{2u_1}{\alpha} - u_0.$$
Considérons par exemple la suite (y_n), on a $\Delta = 4^2 - 4 * 4 = 0$. Il n'y a qu'une racine qui est $r = 2$ (en effet, on peut voir aussi que $x^2 - 4x + 4 = (x - 2)^2$). On a donc $y_n = (an + b)2^n$. Avec $n = 0$ on trouve que $b = y_0 = 1$. Avec $n = 1$ on trouve que $3 = y_1 = (a + b).2$ donc $a = \frac{3}{2} - b = \frac{1}{2}$. Il en résulte que
$$y_n = \left(\frac{1}{2}n + 1\right)2^n.$$
4. **Si Δ est négatif**, l'équation a deux racines complexes conjuguées. Il est aussi possible de résoudre dans ce cas avec une méthode similaire, mais a peine plus compliquée. Nous ne l'étudierons pas ici.

Exercice 30 *Pour chacune des suites suivantes, exprimer u_n en fonction de n.*

1. $u_{n+2} = u_{n+1} + 6u_n$, $u_0 = 1$, $u_1 = -1$.
2. $u_{n+2} = 3u_{n+1} - 2u_n$, $u_0 = 1$, $u_1 = 0$.
3. $u_{n+2} = -2u_{n+1} - u_n$, $u_0 = 1$, $u_1 = 4$.
4. $u_{n+2} = 6u_{n+1} - 9u_n$, $u_0 = -1$, $u_1 = 2$.

Exercice 31 *On considère la suite dite de Fibonacci définie par* $u_{n+2} = u_{n+1} + u_n$ *et* $u_0 = 1$, $u_1 = 1$.

— *Donner* u_n *en fonction de* n.

— *Écrire un algorithme/programme qui calcule efficacement* u_n *en fonction de* n.

Exercice 32 *On considère la fonction suivante :*

```
def tamtam(n):
    if n == 1 or n == 0:
        print "Hello world"
    if n > 1:
        print "Hello world"
        tamtam(n-1)
        tamtam(n-2)
```

1. *Qu'affiche* `tamtam(0)` *?*
2. *Qu'affiche* `tamtam(1)` *?*
3. *Qu'affiche* `tamtam(2)` *?*
4. *Qu'affiche* `tamtam(3)` *?*
5. *On note* u_n *le nombre de fois que la fonction* `tamtam` *affiche de* `Hello world` *en l'appelant sur l'entier* n. *Donner une relation de récurrence sur les* u_n.
6. *Que vaut* u_n *en fonction de* n *?*

Exercice 33 *Une entreprise propose deux type de contrats. Le contrat de* Type1 *avec un salaire annuel de* 23000 *euros et une augmentation de* 500 *euros tous les ans. Le contrat de* Type1 *avec un salaire annuel de* 21000 *euros et une augmentation de* 4% *tous les ans. On note* u_n *le salaire après* n *années dans l'entreprise avec un contrat de* Type1 *et* v_n *celui avec un contrat de* Type2.

1. *Exprimer* u_n *en fonction de* n.
2. *Exprimer* v_n *en fonction de* n.
3. *Calculer* u_5 *et* v_5.
4. *A partir de combien d'années un employée gagne plus avec l'un ou l'autre contrat ?*
5. *A partir de combien d'années un contrat est-il meilleur que l'autre (attention, ça n'est pas la même question que la précédente car il faut compter l'argent gagné en cumulé) ?*

Chapitre 3

Suites : convergence, limites

[1]

3.1 Comportement asymptotique et complexité

Le comportement asymptotique d'une suite est la façon dont elle se comporte (croissance, limite, etc.), pour des valeurs de n grande. On peut distinguer un régime transitoire, de ce qui se passe au début, puis un régime asymptotique, qui se passe de façon stable (on l'espère) à partir d'un certain temps.

On a vu dans le chapitre précédent l'utilisation de suite pour compter le nombre d'opérations (ou d'appels à une fonction spécifique) effectuée par une fonction, en fonction de la valeur d'un paramètre entier. Ces exemples illustrent bien la problématique de complexité mais ce type d'analyse est difficile pour des algorithmes moins simples. Les problèmes suivants se posent notamment :

1. Le nombre d'opérations est parfois difficile à définir précisément ; la relation de récurrence ou de calcul de ce nombre en fonction de n ne peut pas s'exprimer à l'aide d'une jolie formule que l'on sait bien résoudre.
2. Ce qui se passe pour de petites valeurs des entrées n'est pas forcément intéressant et l'on préfère calculer plutôt ce qui se passe quand les entrées deviennent grandes, sans se soucier de l'effet transitoire.
3. Compter avec le même poids chaque opération est une approximation du temps de calcul. Il n'est donc peut-être pas utile de le faire de façon exacte, un ordre de grandeur suffit.
4. Certain (beaucoup) de fonction/algorithmes ont plusieurs paramètres, et pas toujours des entiers. Il est donc difficile de définir une suite du nombre d'opérations en fonction de n, puisqu'il n'y a pas de paramètre d'entrée n.

Les points 1. et 2. ci-dessus vont justifier l'étude asymptotique. Le point 3 va justifier l'étude d'approximations et ce ce qu'on appelle les notations de Landau $O()$, Ω, $\Theta()$ et $o()$, très utilisées dans l'étude des algorithmes.

Pour le point 4., il y a juste un petit ajustement à faire. Pour chaque donnée x, quelle qu'elle soit, on note $\mathsf{size}(x)$ la taille de son codage (en binaire) en machine. Tout algorithme $\mathcal{A}$ prend en entrée une suite $x_1, \dots, x_k$ finie (mais non bornée) de paramètres. La taille de l'entrée de $\mathcal{A}$ est la somme des $\mathsf{size}(x_i)$, c'est-à-dire la taille mémoire totale utilisée pour coder toutes les entrées. On note $C_{\mathcal{A}}(x_1, \dots, x_k)$ le nombre d'opérations effectuées par $\mathcal{A}$ sur l'entrée $x_1, \dots, x_k$. Le nombre de possibilité étant trop important pour une expression simple (sauf s'il n'y a qu'un paramètre

1. http://xkcd.com/1153/

entier, comme dans les exemples vus précédemment), on va agréger tous ces résultats pour une taille d'entrées fixe. On définit alors la complexité en temps $C_{\mathcal{A}}(n)$ de l'algorithme $\mathcal{A}$ comme étant :

$$C_{\mathcal{A}}(n) = \max\{C_{\mathcal{A}}(x_1, \ldots, x_k) \mid \sum_{i=1}^{k} |\mathsf{x_i}| = n\}.$$

Intuitivement, on regarde toutes les entrées possibles de $\mathcal{A}$ dont la taille totale vaut n et on regarde celle qui prend le plus de temps (plus précisément utilise le plus d'opérations élémentaires), c'est ainsi qu'on définit la **complexité dans le pire des cas en temps** de $\mathcal{A}$ [2]. C'est, dans le cadre général, l'étude de $C_{\mathcal{A}}(n)$ qui va nous intéresser.

Il est possible de donner une définition semblable pour l'utilisation d'espace mémoire. Il y a d'autre type de complexités, comme la complexité en moyenne ou la complexité générique, que nous n'étudierons pas.

Considérons par exemple la fonction suivante :

```
def rech0(t):
   for i in range(len(t)):
      if t[i] == 0 :
         return true
   return false
```

Cette fonction recherche si l'élément 0 apparaît dans un tableau. On supposera que le tableau contient des entiers codés sur un espace constant (par exemple 64 bits). Les opérations élémentaires sont le nombre d'incréments du compteur plus le nombre de comparaisons. Comme la taille de chaque entier est constante, on va considérer que la taille d'une entrée d'un tableau est le nombre d'éléments qu'il contient. Le cas le pire, pour une taille n fixée est atteinte lorsque le tableau ne contient par 0 et il y a $2n$ opération. On a donc

$$C_{\texttt{rech0}}(n) = 2 * n.$$

En général, lorsqu'un algorithme prend en entrée un tableau ou une liste, dont les éléments sont codés sur des tailles constantes, on exprime la complexité en fonction du nombre d'éléments.

En général, lorsqu'un algorithme prend en entrée un graphe, on exprime la complexité soit en fonction du nombre de sommets, soit du nombre d'arètes, soit des deux (il faut alors définir des complexités à plusieurs paramètres, ce que nous ne ferons pas pour ne pas alourdir le cours, mais qui ne pose aucun problème technique).

Exercice 34 *On considère la fonction suivante :*

```
def somme(t):
   res=0
   for i in range(len(t)):
      res=res+1
   return res
```

1. *Que fait cette fonction ?*
2. *Quelle est sa complexité ?*

Exercice 35 *On considère la fonction suivante :*

```
def somme(t):
   m=t[0]
   for i in range(len(t)):
      if t[i] > m:
         m=t[i]
   return m
```

1. *Que fait cette fonction ?*
2. *Quelle est sa complexité ?*

Exercice 36 *On considère les fonction suivantes :*

```
def cmp(t):
   for i in range(len(t)):
      for j in range(len(t)):
         if i !=j and t[i]==t[j]:
            return false
   return true
```

2. Pour être plus précis, on définit parfois la complexité pour n comme étant $\max_{k \leq n} C_{\mathcal{A}}(k)$, afin qu'elle soit croissante. Dans ce cours nous n'auront pas cette précision.

```
def cmp2(t):
   for i in range(len(t)):
      for j in range(i+1,len(t)):
         if i !=j and t[i]==t[j]:
            return false
   return true
```

1. *Que font ces fonctions ?*
2. *Quelle sont leurs complexités ?*

3.2 Suites convergentes

L'utilisation des suites convergentes ne sera utile que plus tard pour l'étude de la complexité.

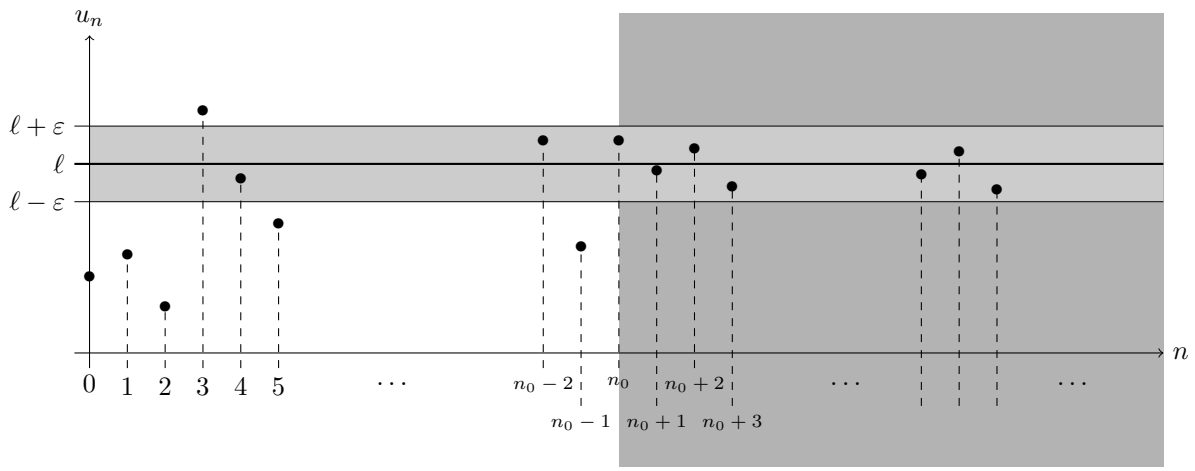
3.2.1 Définitions et exemples

Definition 37 *Une suite (u_n)* **converge** *vers une limite $\ell \in \mathbb{R}$ si, pour tout réel strictement positif ε, il existe un entier n_0 tels que pour tout $n \geq n_0$, $|u_n - \ell| < \varepsilon$. Dans ce cas on note*

$$\lim_{n\to+\infty} u_n = \ell.$$

Une suite qui ne converge pas est dit **divergente**.

Cela signifie que quelque soit la plus petite distance possible ε, si on attend suffisamment longtemps, la suite ne s'éloignera plus jamais de sa limite ℓ de plus de ε, comme cela est illustré sur la figure ci-dessous.



Par exemple la suite définie par $u_n = 1 + \frac{1}{n+1}$ converge vers 1, car en faisant grandir n, on peut rendre $\frac{1}{n+1}$ aussi petit que l'on veut. Il ne serait pas très difficile de montrer formellement (mathématiquement la convergence), mais, en général, la définition est difficile à utiliser. On va donc utiliser différentes techniques indirectes pour étudier la convergence d'une suite.

Mais avant cela, le cas particulier des suites divergentes mais ayant pour limite l'infini sont intéressantes.

Par ailleurs, les suites convergents ont de bonnes propriétés, comme indiqué dans la proposition ci-dessous.

Proposition 38 *Soient (u_n) et (v_n) deux suites convergentes vers respectivement ℓ et m et λ, μ deux réels. Alors $\lambda(u_n) + \mu(v_n)$ converge vers $\lambda\ell + \mu m$. De plus la suite $(u_n v_n)$ converge vers ℓm. Enfin, si (v_n) n'est jamais nulle et si $m \neq 0$, alors la suite $(\frac{u_n}{v_n})$ converge vers $\frac{\ell}{m}$.*

Exercice 39 *On considère la suite définie par $u_n = \frac{n^2-2n+3}{n^3+1}$.*

1. *On considère la suite définie par $v_n = \frac{1}{n} - \frac{2}{n^2} + \frac{3}{n^3}$. Est-elle convergente ? Quelle est sa limite ?*
2. *On considère la suite défîmes par $w_n = 1 + \frac{1}{n^3}$. Est-elle convergente ? Quelle est sa limite ?*
3. *Exprimer u_n en fonction de v_n et w_n.*
4. *Que dire de la convergence de (u_n) ?*

Definition 40 *Une suite (u_n) tend vers plus l'infini [resp. moins l'infini] quand n tend vers plus l'infini si pour tout réel positif [resp. négatif] K, il existe un entier n_0 tels que pour tout $n \geq n_0$, $u_n \geq K$ [resp. $u_n \leq K$]. Dans ce cas on note*

$$\lim_{n\to+\infty} u_n = +\infty \quad \text{[resp. } \lim_{n\to+\infty} u_n = -\infty \text{].}$$

3.2.2 Convergence des suites monotones

Nous allons maintenant voir des techniques pour prouver la convergences d'une suite.

Une suite est **monotone** si elle est croissante ou décroissante. Rappelons que les notions de croissance et décroissances sont données à la définition 23.

Exercice 41 *On considère la suite définie par $u_n = n^2 + n$.*

1. *Que valent u_0, u_1, u_2, u_3 et u_4 ?*
2. *Exprimer u_{n+1} en fonction de n.*
3. *Exprimer $u_{n+1} - u_n$ en fonction de n. Que peut-on en déduire sur (u_n) ?*

Exercice 42 *Dire, en justifiant au mieux, pour chacune des suites suivantes si elle est croissante, décroissante ou ni l'un ni l'autre.*

1. $u_n = 1$,
2. $u_n = n + 1$,
3. $u_n = n^2$,
4. $u_n = (-1)^n$,
5. $u_n = \frac{1}{n+1}$,
6. $u_n = \frac{n}{n+1}$,
7. $u_n = \frac{n}{n^2+1}$,

Exercice 43 *A quelle(s) condition(s) nécessaires et suffisantes (sur la raison et le premier terme) une suite arithmétique est-elle croissante ? décroissante ?*

Exercice 44 *On considère la suite géométrique (u_n) de raison q et premier terme a.*

1. *Exprimer $u_{n+1} - u_n$ en fonction de q, n et a.*
2. *A quelle(s) condition(s) nécessaires et suffisantes (sur la raison et le premier terme) une suite géométrique est-elle croissante ? décroissante ?*

Definition 45 *Une suite (u_n) est* **majorée** *par $M \in \mathbb{R}$ si pour tout n, $u_n \leq M$. Une suite (u_n) est* **minorée** *par $M \in \mathbb{R}$ si pour tout n, $u_n \geq M$.*

Par exemple la suite définie par $u_n = \sin(n)$ est majorée par 1 et minorée par -1. La suite définie par $v_n = 1 + n^2$ est minorée par 1 car n^2 est toujours positif. En revanche elle n'est pas majorée.

Exercice 46

1. *Donner un exemple de suite qui est majorée mais pas minorée.*
2. *Donner un exemple de suite qui n'est pas majorée mais qui ne tend pas vers l'infini.*

Proposition 47 *Soit (u_n) une suite croissante et majorée par un réel M est convergente. De plus sa limite est inférieure à M. Soit (u_n) une suite décroissante et minorée par un réel M est convergente. De plus sa limite est supérieure à M.*

Considérons par exemple la suite définie par $u_0 = 1$ et $u_{n+1} = \frac{u_n}{n}$. On remarque facilement que pour tout n, $u_n \geq 0$ (on peut le démontrer par récurrence facilement) : on divise tout le temps des nombres positifs. Par ailleurs

$$u_{n+1} - u_n = \frac{u_n}{n} - u_n = u_n(\frac{1}{n} - 1) \leq 0,$$

car $u_n \geq 0$ et $\frac{1}{n} - 1 \leq 0$. La suite (u_n) est donc décroissante et minorée, donc elle est convergente.

Exercice 48 *On considère la suite définie par $u_n = \frac{3n+1}{n+2}$.*

1. *Justifier que (u_n) est minorée par 0.*
2. *Justifier que (u_n) est majorée 3.*
3. *Calculer $u_{n+1} - u_n$ en fonction de n.*
4. *Qu'en déduire sur (u_n) ?*

3.2.3 Comparaison de suites

Definition 49 *Soient deux suites u_n et v_n. On dit que (v_n)* **domine** *(u_n), ce que l'on note $(u_n) \leq (v_n)$, si, pour tout entier n, $u_n \leq v_n$.*

Le résultat suivant permet en général de répondre à beaucoup de cas pratiques.

Proposition 50 *Soient (u_n), (v_n) et (w_n) trois suites.*
- *Si $(u_n) \leq (v_n)$ et si $\lim u_n = +\infty$, alors $\lim v_n = +\infty$.*
- *Si $(u_n) \leq (v_n)$ et si $\lim v_n = -\infty$, alors $\lim u_n = -\infty$.*
- *Si $(w_n) \leq (u_n) \leq (v_n)$ et si $\lim v_n = \lim v_n = \ell \in \mathbb{R}$, alors $\lim u_n = \ell$.*

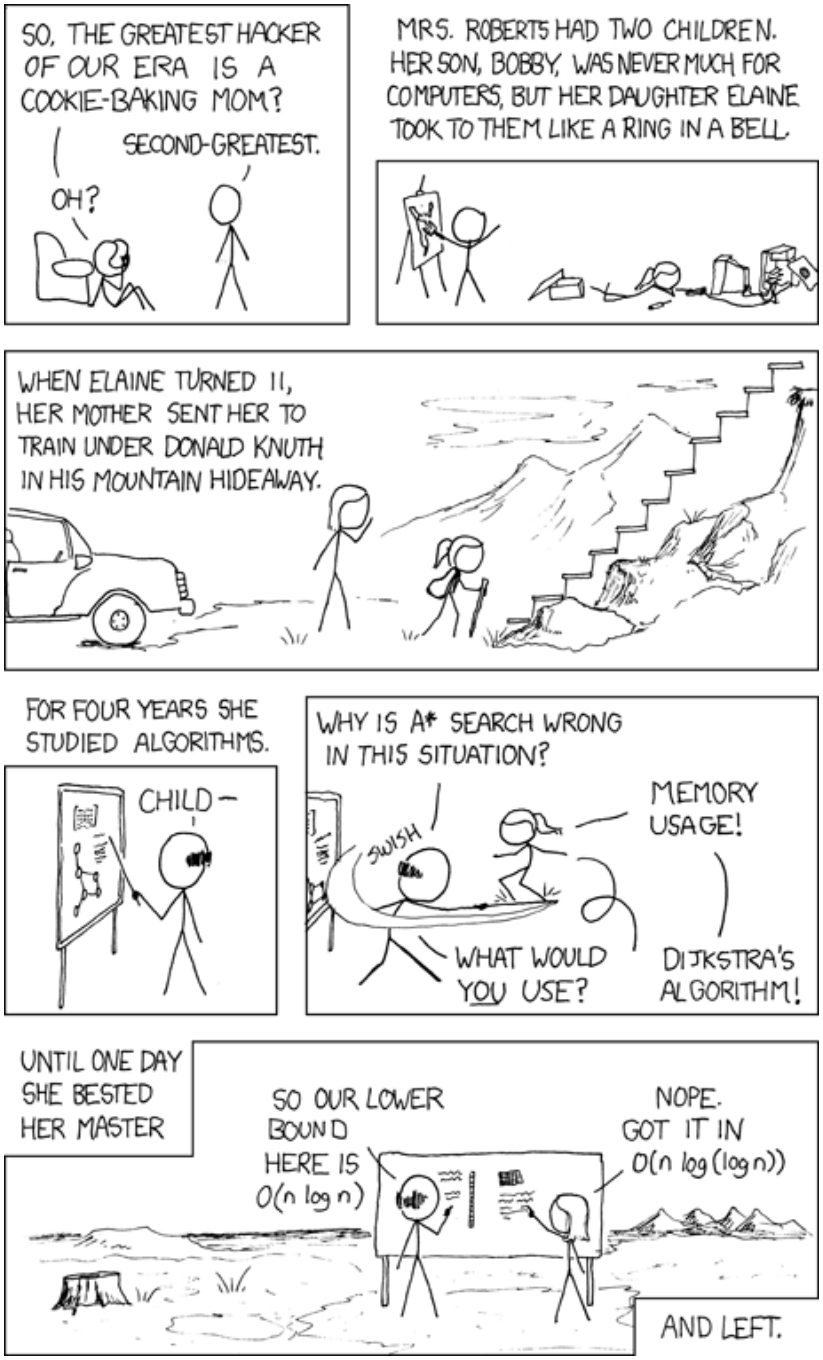
Exercice 51 *Justifier que la suite définie par $u_n = \frac{(-1)^n}{n+1}$ est convergente.*

Exercice 52 *Justifier que la suite définie par $u_n = \frac{n+\sin n}{n+1}$ est convergente.*

Exercice 53 *Quelle est la limite de la suite définie par $u_n = n + \sqrt{1+\cos n}$.*

Chapitre 4

Notations de Landau



[1]

1. http://xkcd.com/342/

4.1 Introduction

Le notations de Landau permettent de faire de faciliter le calcul d'ordres de grandeur de suites (mais aussi de fonctions, ce que nous ne verrons pas). La plus utilisée en informatique est le $O()$. Les plus utilisées en mathématiques sont sûrement le $o()$ et le $\sim$. il y a aussi $\Omega()$ et $\Theta()$. Nous n'étudierons que $O()$ dans ce cours.

4.2 Notation $O()$

Definition 54 *Soient (u_n) et (v_n) deux suites à valeurs positives. On écrit $u_n = O(v_n)$ (ou $u_n \in O(v_n)$) s'il existe un entier n_0 et une constante K tels que pour tout $n \geq n_0$, $u_n \leq Kv_n$.*

Intuitivement cela signifie que que pour n assez grand, u_n sera majorée par une constante fois v_n. Par exemple $3n^2 - n + 5 = O(n^2)$ car pour $n \geq 5$, $3n^2 - n + 5 \leq 3n^2$ (ici on peut prendre $K = 3$).

Exercice 55 *Que peut-on dire de la suite (u_n) si $u_n = O(1)$?*

Exercice 56 *Soient p et k deux constantes. Quand a-t-on $n^k = O(n^p)$?*

Exercice 57 *Soient (u_n), (v_n), (x_n) et (y_n) trois suites à valeurs positives telles que $u_n = O(v_n)$ et $x_n = O(y_n)$.*
- *Justifier que $u_n + x_n = O(v_n + y_n)$.*
- *Justifier que $u_n x_n = O(v_n y_n)$.*
- *On suppose de plus pour cette question uniquement que $v_n = O(x_n)$. Justifier que $u_n = O(y_n)$ et $u_n = O(x_n)$.*
- *On suppose maintenant que $v_n = O(y_n)$. Justifier que $u_n = O(y_n)$ et que $u_n + x_n = O(y_n)$.*

La proposition suivante est très importante en algorithmique. La complexité d'un alpgorithme est souvent exprimée dans cette échelle, c'est-à-dire qu'on dit que le nombre d'opération est en $O(r^n n^\alpha (\log n)^\beta)$. Parfois on a besoin d'autres fonctions, mais cela permet de couvrir, en pratique, la plupart des cas rencontrés.

Proposition 58 *On a*

$$r_1^n n^{\alpha_1} (\log n)^{\beta_1} = O\left(r_2^n n^{\alpha_2} (\log n)^{\beta_2}\right),$$

si
- *$r_1 < r_2$, ou*
- *$r_1 = r_2$ et $\alpha_1 < alpha_2$, ou*
- *$r_1 = r_2$ et $\alpha_1 = alpha_2$ et $\beta_1 < beta_2$.*

Un algorithme dont la complexite est en $O(f)$ sera dit meilleur qu'un algorithme dont la compelxité est en (h) si $f = O(h)$. En fait, il y a dans cette assertion des sous-entendus important, à savoir que f et h sont les approximations (dans l'échelle utilisée) les plus précises des complexités. Par ailleurs, il ne faut pas oublier qu'il s'agit de résultats asymptotiques à une constante mutliplicative près.

Exercice 59 *Classer les différentes complexités par ordre croissant d'éfficacité. $O(n \log n)$, $O(n^2 \log n)$, $O(n \log \log n)$, $O(n \log^3 n)$, $O(n)$ et $O(n^2)$.*

Exercice 60 *Les assertions suivantes sont elles vraies ou fausses, pourquoi ?*
1. $3^n = O(2^n)$,
2. $\log 3^n = O(\log 2^n)$.

4.3 Exercices

Exercice 61 *1. Justifier brièvement à quoi sert le code suivant :*

```
int Recherche(int tab[], int n, int k){
  int i;
  for(i=0;i<n;i++){
    if(tab[i]==k){
      return 1;
    }
  }
  return 0;
}
```

2. Exprimer la complexité de cet algorithme en fonction du paramètre `n`.
3. On considère la fonction suivante :

```
int RechercheD(int* tab, int debut, int fin, int k){
  int new;
  printf("deb = %d, fin = %d \n", debut, fin);
  new=(fin+debut)/2;
  if (tab[new]==k){
    return 1;
  }
  if(new==debut || new == fin){
    return 0;
  }
  if (tab[new] < k){
    return RechercheD(tab,new,fin,k);
  }
  if (tab[new] > k){
    return RechercheD(tab,debut,new,k);
  }
}
```

Expliquer le déroulement de la fonction `RechercheD(t,0,9,41);` *sur le tableau*
`t= [0, 3, 7, 11, 12, 13, 32, 41, 58, 110]`
A quoi sert cette fonction sur un tableau trié ? Exprimer sa complexité en fonction de la taille du tableau d'entrée.
4. On dispose d'un tableau de n entiers. On dispose aussi d'une liste de $O(n)$ entiers. On desire savoir, pour chacun de ces $O(n)$ entiers s'il figure dans le tableau. Faut-il utiliser la fonction `Recherche` *ou la fonction* `RechercheD` *(après avoir trié le tableau et en supposant que l'on peut trier en temps $O(n \log n)$) ? (on ne demande pas d'algorithme).*
5. Même question, mais la liste contient $O(\log \log n)$ entiers.
6. Même question, mais la liste contient $O(\log n)$ entiers.

Exercice 62 *1. En supposant que l'on dispose de deux algorithmes, l'un ayant une complexité en $O(n^{\alpha_1} \log^{\beta_1} n)$, l'autre en $O(n^{\alpha_2} \log^{\beta_2} n)$. Dire en fonction de α_1, α_2, β_1 et β_2 lequel on doit utiliser ?*

2. On considère que l'on dispose maintenant de trois structures de données S1, S2 *et* S3 *permettant de mémoriser des entiers. On considère aussi trois types de requêtes,* R1, R2 *et* R3 *que l'on peut effectuer. La complexité de chacune de ces requêtes pour chaque structure de données est exprimée dans le tableau suivant :*

	S1	S2	S3
R1	$O(n)$	$O(n)$	$O(n^2)$
R2	$O(n \log n)$	$O(n^2)$	$O(n)$
R3	$O(n^3)$	$O(n^2)$	$O(n \log^2 n)$

2.1 On considère que l'on va faire $n \log n$ requêtes R1, *$\log n$ requêtes* R2 *et $\log n$ requêtes* R3 *sur n entiers. Donner, pour chacune des structures* S1, S2 *et* S3, *la complexité totale de toutes ces requêtes. On exprimera cette complexité sous la forme $O(n^{\alpha} \log^{\beta} n)$. Quelle structure vaut-il mieux utiliser ?*

2.1 On considère que l'on va faire k requêtes R1, *k requêtes* R2 *et k requêtes* R3 *sur n entiers. Donner, pour chacune des structures* S1, S2 *et* S3, *la complexité totale de toutes ces requêtes. On exprimera cette complexité en fonction de n et k. Quelle structure ne faut-il pas utiliser ? Que peut-on dire des deux autres ?*