Complexité d'un algorithme

Trois questions à se poser quand on fabrique un algorithme :
- est-ce qu'il donne un résultat ? $\rightsquigarrow$ terminaison ;
- est-ce qu'il donne le/un bon résultat ? $\rightsquigarrow$ validité ;
- est-ce qu'il donne le résultat en temps raisonnable ? $\rightsquigarrow$ complexité.

1 Calculs des nombres de Fibonacci

On cherche à calculer les nombres de Fbonacci $(F_n)_{n\geq 1}$ définis par :

$$F_0 = F_1 = 1, \quad \forall n \in \mathbb{N},\ F_{n+2} = F_{n+1} + F_n.$$

1.1 Calcul récursif

La méthode récursive est élégante, mais on va voir qu'elle n'est pas très efficace. Voici le code correspondant :

Xcas

```
f(n) := {
   if (n<2) return 1 ;
   return f(n-1) + f(n-2) ;
}
```

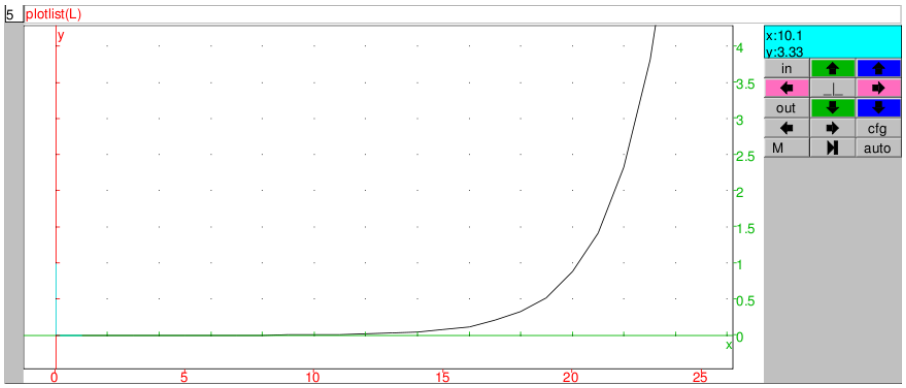
1. Tester le calcul sur quelques valeurs.
2. L'instruction `time(f(n))` calcule le temps $t(n)$ mis pour calculer $f(n)$. Dessiner les points de coordonnées $(n, t(n))$ pour n variant dans un intervalle « intéressant ». Faire un ajustement exponentiel.
3. On note a_n le nombre d'appels à la fonction `f` effectués pendant le calcul de $f(n)$ et s_n le nombre d'additions nécessaires. Par exemple, on a : $a_0 = a_1 = 0$, $a_2 = 1$, $s_0 = s_1 = 0$, $s_1 = 1$.
   Donner des relations de récurrences déterminant les suites (a_n) et (s_n). Préciser et interpréter les mesures du temps de calcul.

Une résolution

2. La fonction `temps_f` renvoie le couple (n, t_n), puis on engendre une liste de couples que l'on trace :

Xcas

```
temps_f(n):= {
   local s ;
   s := time(f(n)) ;
   return [n,s];
} ;
L := [seq(temps_f(n),n,0,20)] ;
plotlist(L) ;
```



On a l'impression d'avoir tracé la courbe représentative de l'exponentielle ! Pour vérifier l'adéquation, on calcule et on trace les points $(n, \ln t_n)$, puis on fait un ajustement linéaire sur les derniers points (les erreurs paraissant trop importantes pour les premiers points) :

Xcas

```
N := seq([n,ln(L[n][1])],n,1,20) ;
plotlist(N) ;
linear_regression(N[10..20]) ;
```

3. Lors du calcul de $f(n)$, on appelle une fois la fonction lorsqu'on veut calculer $f(n-1)$, calcul qui engendre a_{n-1} appels ; on l'appelle également $1 + a_{n-2}$ fois pour le calcul de $f(n-2)$. De même, on fait s_{n-1} sommes pour calculer $f(n-1)$, s_{n-2} pour calculer $f(n-1)$ et une de plus pour calculer $f(n)$. D'où, pour $n \geq 2$:

$$a_n = a_{n-1} + 1 + a_{n-2} + 1, \quad s_n = s_{n-1} + s_{n-2} + 1.$$

Les suites $(a_n + 2)_{n\geq 0}$ et $(s_n + 1)_{n\geq 0}$ satisfont à la même définition par récurrence que la suite de Fibonacci, aux premiers termes près. Aussi, elles sont équivalentes à $C\varphi^n$, où φ est le nombre d'or et C une constante convenable (voir ci-dessous).

4. Si on estime que l'ordinateur met un temps constant pour chaque appel à f (à peu près raisonnable, sauf qu'on néglige complètement les gestions de mémoire causée par l'empilement des boucles) et chaque somme (raisonnable, du moins tant que les nombres en jeu ne sont pas trop grands), on s'attend à ce que le temps de calcul soit une combinaison de a_n et s_n, du moins tant que les données ne sont pas trop grandes. Cela donnerait un temps de calcul de $f(n)$ en $C\varphi^n$, ce qui est confirmé pour les valeurs « à échelle humaine ».

1.2 Calcul par boucle simple

1. Proposer une méthode par itération simple pour calculer les nombres de Fibonacci. À quel temps de calcul s'attend-on ?
2. Implémenter, mesurer le temps de calcul, dessiner un nuage de points et vérifier ou infirmer l'hypothèse précédente.

Une résolution

1. On garde en mémoire deux entiers `a` et `b`, on calcule leur somme `c`, on remplace `a` par `b` et `b` par `c` ; on recommence autant de fois que nécessaire.

   Pour calculer le n^{e} nombre de Fibonacci, on fait une boucle de longueur $n-1$ et, dans chaque boucle, on fait une addition. Pour autant que les données ne soient pas trop grandes, on va estimer que l'addition prend un temps constant, on s'attend donc à un temps de calcul linéaire en n.

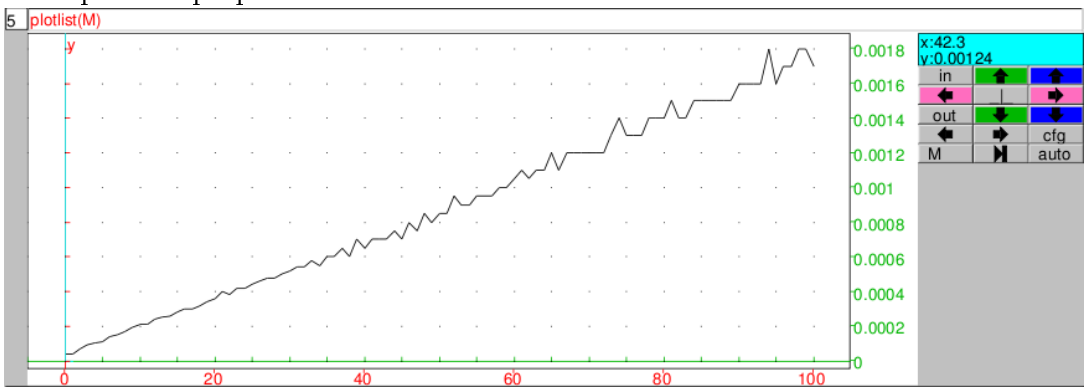
2. Avec Xcas, cela peut donner ceci :

Xcas

```
g(n) := {
    local a,b,c,i,r ;
    if(n<2) r := 1;
    else {
        a := 1 ; b := 1
        for(i:=1;i<n;i++) {
            c := a+b ;
            a := b ;
            b := c ;
        }
        r := c ;
    }
    return r ;
}

temps_g(n):= {
    local s ;
    s := time(g(n)) ;
    return [n,s];
} ;
M := seq(temps_g(n),n,0,80) ;
plotlist(M) ;
```

L'allure de la courbe des mesures est conforme aux attentes, avec des oscillations que l'on ne cherchera pas à expliquer :



1.3 Calcul matriciel

1. Pour n entier naturel, on considère la matrice-ligne $X_n = (F_{n+1} \quad F_n)$. Donner une formule de récurrence matricielle et retrouver ainsi la formule de Binet exprimant F_n en fonction de n et du nombre d'or.
2. En déduire une méthode bien plus efficace que les précédentes pour le calcul de (F_n) par le calcul d'une puissance de matrice (sans utiliser les opérations matricielles prédéfinies).

Une résolution

1. Pour n entier naturel, on considère la matrice-ligne :

$$X_n = (F_{n+1} \quad F_n).$$

On se convainc sans peine que pour tout n, on a :

$$X_{n+1} = X_n A, \quad \text{où } A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}.$$

Il en résulte classiquement que $X_n = X_0 A^n$. Le calcul des premières valeurs de A^n montre que ses coefficients sont eux-mêmes des nombres de Fibonacci. Une récurrence facile donne plus précisément (avec $F_{-1} = 0$) :

$$\forall n \in \mathbb{N}, \quad A^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}.$$

Pour le calcul théorique, la méthode habituelle consiste à diagonaliser A, etc. Passons.

2. La partie délicate, c'est la stratégie *a priori* pour calculer une puissance en minimisant le nombre de produits à effectuer, sans se préoccuper de l'endroit où on fait les calculs (matrices, nombres entiers, réels...). L'approche naïve, fondée sur la relation $A^n = A \cdot A^{n-1}$ conduit à effectuer $n-1$ produits pour une puissance n^{e}. Une méthode plus rapide consiste à procéder de la façon récursive suivante :
   – si n est pair, calculer A^n comme le carré de $A^{n/2}$.
   – si n est impair, calculer A^n comme le produit de A et du carré de $A^{(n-1)/2}$.
   En code, cela donne :

Xcas

```
pr(A,B) := {
 local C;
 C :=
 [[A[0][1]*B[1][0]+A[0][0]*B[0][0] ,A[0][1]*B[1][1]+A[0][0]*B[0][1]] ,
 [ A[1][1]*B[1][0]+A[1][0]*B[0][0] ,A[1][1]*B[1][1]+A[1][0]*B[0][1]]];
 return C ;
} ;
ca(A) := {
    return pr(A,A) ;
} ;
Id := [[1 ,0] ,[0 ,1]]
pu(A,n) := {
    local m ;
    if(n==0) return Id ;
    else {
        if(n mod 2 ==0) return ca(pu(A,n/2)) ;
        else return pr(A,ca(pu(A,(n-1)/2))) ;
    }
} ;
```

Exercice amusant : proposer une alternative non récursive de ce calcul de puissances (voir par exemple le début du cours d'Yves Robert).

2 Algorithme de Gauss

Évaluer le nombre d'opérations (additions/soustractions et multiplications/divisions) que l'on effectue en résolvant un système linéaire à n équations et n inconnues générique (ayant une unique solution) avec l'algorithme de Gauss.
Pourquoi est-il raisonnable de négliger les comparaisons (pour trouver le plus grand pivot dans chaque colonne) et les permutations de lignes (pour placer le pivot à la bonne place) ?

Une résolution

1. Soit $A = (a_{ij})_{i,j=1,\dots,n}$ la matrice du système. La première étape de l'algorithme de Gauss consiste à manipuler les lignes pour rendre la matrice triangulaire supérieure. Pour cela, on suppose que $a_{11} \neq 0$ –sinon, $n-1$ comparaisons et une permutation de deux lignes permettent de remplacer le pivot a_{11} par le coefficient de valeur absolue maximale sur la première colonne. Soit i un entier compris entre 2 à n. On remplace la i^{e} ligne L_i du système par $L_i - \rho_i L_1$, où $\rho_i = a_{i1}/a_{11}$. Pour cela, on effectue une division pour calculer ρ_i, n multiplications et autant d'additions pour la matrice et le second membre, soit $(n+1)$ multiplications et n additions. Comme il y a $n-1$ lignes, cela fait, pour cette étape, n^2-1 multiplications et n^2-n additions. On s'est ramené à un système $(n-1)\times(n-1)$. Pour cette première étape, on trouve donc :
   – nombre de multiplications : $\sum_{k=1}^{n}(k^2-1) = n(n-1)(2n+5)/6 \sim n^3/3$,
   – nombre d'additions : $\sum_{k=1}^{n} k(k-1) = n(n-1)(n+1)/3 \sim n^3/3$.

On a à présent un système triangulaire à résoudre. La dernière variable s'obtient avec 1 division. L'avant-dernière s'obtient avec 1 multiplication, 1 soustraction et 1 division. Pour $k \geq 1$, le calcul de la variable numéro $n-k$ nécessite $k-1$ multiplications, $k-1$ additions et 1 division, soit, en tout, k multiplications et $k-1$ additions. Dans cette étape, on trouve donc :
– nombre de multiplications : $\sum_{k=1}^{n} k = k(k+1)/2 \sim n^2/2$,
– nombre d'additions : $\sum_{k=1}^{n}(k-1) = n(n-1)/2 \sim n^2/2$.
En ajoutant le nombre d'opérations des deux étapes et en supposant n grand, on constate que l'algorithme de Gauss demande environ $n^3/3$ additions et multiplications ; en simplifiant un peu : *l'algorithme de Gauss pour une matrice $n \times n$ demande $O(n^3)$ opérations.*

Remarque On a bien fait de négliger les comparaisons, car il en faut au maximum $n-1$ pour la première colonne, $n-2$ pour la deuxième, etc., soit $O(n^2)$ en tout. Quant aux permutations de lignes, il y en a $O(n)$ au maximum.

2. **Un exemple standard.** On veut résoudre numériquement l'équation différentielle

$$\frac{\mathrm{d}^2 u}{\mathrm{d}x^2} = f,$$

où f (donnée) et u (inconnue, nulle en 0 et 1) sont deux fonctions de $[0,1]$ dans . On fixe un entier strictement positif n et on subdivise $[0,1]$ en n intervalles de largeur $h = 1/n$. Pour un entier i compris entre 0 et n, on cherche une approximation u_i de la valeur de u en $x_i = i/n$. Pour n assez grand, on utilise l'approximation (pourquoi ?) :

$$\frac{\mathrm{d}^2 u}{\mathrm{d}x^2}(x_i) \simeq \frac{u(x_{i+1}) + u(x_{i-1}) - 2u(x_i)}{h^2}.$$

L'équation différentielle discrétisée se traduit donc par un système $n \times n$, dont on montre qu'il a une unique solution.

3. À présent, on s'intéresse à des fonctions de trois variables définies sur le cube $[0,1]^3$, satisfaisant l'équation du laplacien :

$$\Delta u = f, \quad \text{où } \Delta u = \frac{\mathrm{d}^2 u}{\mathrm{d}x^2} + \frac{\mathrm{d}^2 u}{\mathrm{d}y^2} + \frac{\mathrm{d}^2 u}{\mathrm{d}z^2}.$$

La même méthode conduit naturellement à un système $n^3 \times n^3$. Si on coupe chaque intervalle de la grille en $n = 100$, résoudre le système par l'algorithme de Gauss demanderait un nombre d'opérations proportionnel à $(100^3)^3 = 10^{18}$: inconcevable !

3 Algorithme d'Euclide (2)

Rappeler l'algorithme d'Euclide (on suppose disposer des opérations arithmétiques et de la partie entière)... Évaluer, en fonction des valeurs des paramètres entrés, le nombre *maximal* de divisions euclidiennes qu'il faut effectuer.

Une résolution

Notons (a_0, a_1) les entiers dont on cherche le pgcd. On les suppose positifs et « dans le bon ordre » :

$$a_0 \geq a_1.$$

L'algorithme définit deux suites finies (a_i), (q_i) de proche en proche par :

$$\begin{array}{rcll} a_0 & = & a_1 q_1 + a_2, & 0 \leq a_2 < a_1 \\ a_1 & = & a_2 q_2 + a_3, & 0 \leq a_3 < a_2 \\ & & \cdots & \\ a_{r-2} & = & a_{r-1} a_{r-1} + a_r & 0 \leq a_r < a_{r-1} \\ a_{r-1} & = & a_r q_r + 0 & a_{r+1} = 0 \end{array}$$

On appellera r le nombre d'étapes : c'est le nombre de lignes du tableau, donc le nombre de divisions euclidiennes. L'idée, c'est que le nombre d'étape est maximal lorsque la suite (a_i) décroît le moins possible, c'est-à-dire lorsque les quotients successifs sont aussi petits que possible. On a dans ce cas $a_{i+2} = a_i - a_{i+1}$, ce qui ressemble à la suite de Fibonacci.
Pour formaliser cette idée, on commence par renuméroter $(a_0, \ldots, a_r, a_{r+1} = 0)$ en commençant par la fin, ce qui revient à poser :

$$\forall k \in \{0, \ldots, r+1\}, \quad u_k = a_{r+1-k}.$$

Pour $i \in \{1, \ldots, r\}$, on a :

$$a_{i-1} = q_i a_i + a_{i+1},$$

ce qui s'écrit, pour $k \in \{1, \ldots, r\}$:

$$u_{k+1} = q_{r+1-k} u_k + u_{k-1}.$$

La remarque-clé, à savoir que l'entier q_{r+1-k} vaut au moins 1, conduit à :

$$\forall k \in \{1, \ldots, r\}, \quad u_{k+1} \geq u_k + u_{k-1}.$$

Ceci conduit à comparer (u_k) à la suite finie (f_k) définie par :

$$f_0 = 0, \quad f_1 = 1, \quad \forall k \geq 1,\ f_{k+1} = f_k + f_{k-1}.$$

C'est bien sûr la suite de Fibonacci. Si on préfère éviter le décalage, en notant $F_k = f_{k+1}$, la suite (F_k) estla « vraie » suite de Fibonacci : $(1, 1, 2, 3, 5, 8, \ldots)$.
On a : $u_0 = 0 = f_0$ et $u_1 \geq f_1$ (car $u_1 = a_r$ n'est pas nul), donc par récurrence

$$\forall k \leq r+1, \quad u_k \geq f_k.$$

En particulier, on obtient $u_{r+1} \geq f_{r+1}$, soit $a_0 \geq F_r$, et de même : $a_1 \geq F_{r-1}$.
En d'autres termes, *le nombre d'étapes de l'algorithme r d'Euclide est, au pire, l'indice du plus grand nombre de Fibonacci inférieur au plus grand des deux entiers.* Si $a_0 = F_r$ et $a_1 = F_{r-1}$, on se convainc que le nombre d'étapes est exactement r (la relation qui définit les nombres de Fibonacci est la division euclidienne de F_k par F_{k-1} !).
Connaissant la formule de Binet pour F_n, on en déduit aussi que

$$r \leq \log_\varphi a_0 + \text{Cste}.$$

Ceci traduit que l'algorithme d'Euclide est *efficace*, puisqu'il est linéaire en la taille des données (qui est le nombre de chiffres, soit logarithmique en a_0 et a_1).