Escapade algorithmique avec Fibonacci

Arnaud de Saint Julien
desainta@yahoo.fr

Introduction

La suite d'entiers $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$ où chaque terme s'obtient en additionnant les deux précédents porte le nom de suite de Fibonacci, en hommage à Leonardo Fibonacci mathématicien du XIIIe siècle qui l'avait utilisée pour modéliser l'évolution d'une population de lapins.

En langage moderne, cette suite que l'on notera $(F_n)_n$ est définie par ses deux premiers termes $F_0 = 0$, $F_1 = 1$ et la relation de récurrence

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

Ce document propose une escapade algorithmique avec les nombres de Fibonacci. Nous aborderons des thèmes au coeur du programme commun d'informatique des classes préparatoires, notamment : algorithme d'Euclide, récursivité, complexité, exponentiation rapide. Tous les algorithmes seront écrits en «pseudo-code» puis traduits en langage Maple.

1 Premiers calculs des nombres de Fibonacci

Imaginons un instant que nous soyons M. Fibonacci et que nous ayons besoin de calculer F_{50}, (par exemple pour estimer le nombre de lapins au bout de 50 mois). Ce n'est pas difficile, il suffit d'effectuer les additions successives $0+1=1,\ 1+1=2,\ 1+2=3,\ 3+2=5,\ 5+3=8\ldots$, mais c'est long puisque nous devons faire 49 additions.

Aujourd'hui nous pouvons demander à un ordinateur d'effectuer ces additions pour nous, il faut pour cela disposer d'un algorithme puis le traduire dans un langage interprétable par l'ordinateur : c'est la phase de programmation. Nous allons présenter dans cette section, quelques premiers algorithmes qui prennent en entrée un entier naturel n et renvoient la valeur du nombre de Fibonacci F_n.

Rappelons auparavant quelques éléments de syntaxe :
- le symbole d'affectation : par exemple j'affecte la variable u de la valeur 0 est notée $u \leftarrow 0$ en pseudo-code, et s'écrit «`u :=0`» en Maple.
- si `f` est une table ou une liste, `f[k]` désigne son terme d'indice k.

1.1 Avec la formule de récurrence

Un premier algorithme (écrit en pseudo-code puis traduit en Maple) que l'on nomme `fibotable` consiste à construire pas à pas une table contenant les nombres $F_0, F_1, \ldots, F_n$ dans une variable nommée `f`.

algorithme : `fibotable`
Entrées : $n \in \mathbb{N}$
Résultat : F_n
Variables : f une table,k
  $f[0] \leftarrow 0,\ f[1] \leftarrow 1$
  Pour k de 2 à n, faire
    $f[k] \leftarrow f[k-1] + f[k-2]$
  Fin du pour
  Renvoyer $f[n]$

```
> fibotable:=proc(n::nonnegint)
> local f,k;
> f[0]:=0;f[1]:=1;
> for k from 2 to n do
> f[k]:=f[k-1]+f[k-2];
> od;
> RETURN(f[n]);
> end;
```

On le lance,

```
> fibotable(50);
```

12586269025

Le nombre F_{50} vaut donc 12586269025. Le résultat est instantané.

L'utilisation de la fonction `time` de Maple permet de mesurer les temps de calcul. Par exemple, l'instruction suivante montre que `fibotable` calcule F_{20000} en 1.563 secondes.

```
> time(fibotable(20000));
```

1.563

Cet algorithme est assez facile à rédiger et à lire car il est très proche de la définition par récurrence de la suite (F_n). Il effectue $n-1$ additions pour calculer F_n comme nous le ferions à la main. Il admet toutefois un inconvénient assez important, il nécessite l'affectation de nombreuses variables. En effet, par exemple pour calculer F_{50}, l'algorithme `fibotable` a utilisé 52 variables de type entier qui sont $f[0], f[1], \ldots, f[50]$ et k la variable du compteur «pour». Cela pose lorsque n devient grand de sérieux problèmes d'occupation de mémoire et donc aussi d'efficacité. D'ailleurs, Maple a de gros soucis lorsqu'on lui demande `time(fibotable(80000))`.

```
> time(fibotable(80000)):
```

```
System error, ran out of memory
```

Voici un deuxième algorithme nommé `fibo` assez proche du premier mais qui gomme le problème des trop nombreuses affectations. En contre partie, il est un plus difficile à rédiger.

algorithme : `fibo`
Entrées : $n \in \mathbb{N}$
Résultat : F_n
Variables : u, v, s, k
  Si $n = 0$ ou $n = 1$ alors
    Renvoyer n
  Fin du si
  $u \leftarrow 0, v \leftarrow 1$
  Pour k de 1 à $n-1$, faire
    $s \leftarrow u + v$
    $u \leftarrow v$
    $v \leftarrow s$
  Fin du pour
  Renvoyer s

```
> fibo:=proc(n::nonnegint)
> local u,v,s,k;
> if n=0 or n=1 then RETURN(n);
> fi;
> u:=0;v:=1;
> for k from 0 to n-2 do
> s:=u+v;
> u:=v;
> v:=s;
> od;
> RETURN(s);
> end:
```

On le teste avec Maple,

```
> fibo(50);
```

12586269025

Cette fois-ci, `fibo` n'utilise que 4 variables u, v, s et k au lieu de $n+2$ quelque soit la valeur de n, sa complexité spatiale est bien meilleure que `fiboliste`. Même s'il effectue lui aussi $n-1$ additions pour calculer F_n, `fibo` est plus efficace que `fiboliste` comme on peut l'observer sur le tableau suivant :

n	100	1000	10000	20000	50000	80000	10^6
temps mis par `fibotable(n)` en s	0	0	0.344	1.563	23.125	∞	∞
temps mis par `fibo(n)` en s	0	0	0.187	0.656	3.750	9.563	14.531

1.2 Les limites numériques de la formule de Binet

On connait [1]depuis le XIXe siècle la formule de Binet qui donne l'expression des nombres de Fibonacci à l'aide du nombre d'or $\phi = \dfrac{1+\sqrt{5}}{2}$:

$$F_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right].$$

Cette formule est remarquable :
– elle donne l'expression d'une suite d'entiers à l'aide du célèbre nombre d'or qui est irrationnel,

1. De nos jours, un résultat de cours sur les suites récurrentes linéaires d'ordre 2 permet d'obtenir rapidement le résultat. Les racines réelles de l'équation caractéristique $r^2 - r - 1 = 0$ sont ϕ et $\overline{\phi}$. Ainsi F_n est combinaison linéaire des termes géométriques ϕ^n et $\overline{\phi}^n$, c'est-à-dire il existe des réels a et b tels que pour tout n, $F_n = a\phi^n + b\overline{\phi}^n$. On détermine enfin les réels a et b avec les conditions initiales $F_0 = 0$ et $F_1 = 1$.

– elle permet de calculer F_n sans à avoir à calculer les termes précédents F_{n-1} et F_{n-2},
– elle permet d'obtenir un équivalent simple de F_n pour n au voisinage de $+\infty$. En effet si on pose $\overline{\phi} = \dfrac{1-\sqrt{5}}{2}$, comme $|\phi| > 1$ et $\left|\overline{\phi}\right| < 1$, la suite géométrique $(\phi^n)_n$ tend vers $+\infty$ tandis que $(\overline{\phi}^n)_n$ tend vers 0, donc F_n est équivalent à $\dfrac{\phi^n}{\sqrt{5}}$.

Comme Maple est un logiciel de calcul formel, nous pouvons lui demander de développer (avec `expand`) l'expression de la formule de Binet pour récupérer la valeur de l'entier F_n.

```
> phi:=(1+sqrt(5))/2: phib:=(1-sqrt(5))/2:
> g:=n->expand((phi^n-phib^n)/sqrt(5)):
> g(50);
```

12586269025

C'est très simple à rédiger mais bien moins efficace que l'algorithme `fibo` comme vous pouvez le constater

n	100	500	1000	2000
temps mis par `g(n)` en s	0	0.657	4.5	28.437

Une autre possibilité pourrait être alors de faire du calcul approché avec la fonction `evalf` (diminutif de «evaluate using floating-point arithmetic») de Maple. Comme F_n est un entier, il suffit de demander une valeur approchée de F_n par exemple à 0.5 près puis de considérer l'entier le plus proche (`round` en Maple).

Rappelons que par défaut, `evalf` calcule avec 10 chiffres significatifs. On peut préciser le nombre N de chiffres signicatifs de x, en tapant `evalf(x,N)`. Par exemple `evalf(Pi,3)` renvoie $3,14$.

Testons :

```
> phi:=(1+sqrt(5))/2: phib:=(1-sqrt(5))/2:
> f:=n->(phi^n-phib^n)/sqrt(5):
> round(evalf(f(50)));
```

12586269120

Oups ! Il nous dit que F_{50} vaut 12586269120 au lieu de 12586269025. Les erreurs d'arrondies cumulées ont été trop importantes. Si l'on augmente le nombre de chiffres significatifs (à partir de 12), la bonne valeur de F_{50} est renvoyée. Toute la difficulté est de connaître le degré de précision à adopter pour obtenir les bons résultats. Nous n'aborderons pas dans ce texte cette problématique et nous contenterons des observations. Le lecteur intéressé pourra consulter le premier chapitre du livre [3].

2 La rencontre improbable d'Euclide et de Fibonacci

L'algorithme d'Euclide est un des plus anciens algorithmes de calcul. Il permet de calculer le «Plus Grand Commun Diviseur» (PGCD) de deux nombres entiers. Il est remarquable par sa simplicité (des divisions successives), la diversité de ses applications mais aussi son efficacité. Dans cette section, nous allons «mesurer» cette efficacité (on parle de complexité d'algorithme) à l'aide des nombres de Fibonacci.

Rappelons tout d'abord l'algorithme d'Euclide que nous avons nommé `euclide` qui renvoie le pgcd de deux entiers naturels a et b. Il repose sur les deux propriétés suivantes :
– $\text{pgcd}(a,0) = a$.
– si r est le reste de la division euclidienne de a par b, alors $\text{pgcd}(a,b) = \text{pgcd}(b,r)$.

algorithme : `euclide`
Entrées : $a, b \in \mathbb{N}$
Résultat : $\text{pgcd}(a,b)$
Variables : u, v, r
  $u \leftarrow a$
  $v \leftarrow b$
  Tant que $v \neq 0$, Faire
    $r \leftarrow$ le reste de la divison de u par v
    $u \leftarrow v$
    $v \leftarrow r$
  Fin du Tant que
  Renvoyer u

```
> euclide:=proc(a::nonnegint,b::nonnegint)
>    local u,v,r;
>    u:=a;
>    v:=b;
>    while v<>0 do
>       r:=irem(u,v);
>       u:=v;
>       v:=r;
>    od;
>    RETURN(u);
> end:
```

Notre programme `euclide` s'écrit à l'aide d'une boucle «tant que», c'est une version dite *itérative*. Nous verrons à la section suivante une version dite *récursive*.

La proposition fondamentale suivante permet d'obtenir une majoration du nombre de divisions euclidiennes nécessaires. Les nombres de Fibonacci font leur apparition.

Proposition 1 *Soit a et b deux entiers tels que $a > b \geqslant 1$. Si `euclide(a,b)` effectue $k \geqslant 1$ divisions euclidiennes, alors $a \geqslant F_{k+2}$ et $b \geqslant F_{k+1}$.*

Preuve :
Par récurrence sur l'entier k.

L'initialisation est pour $k = 1$. On a bien $b \geqslant 1 = F_2$ et $a > b$ donc $a \geqslant 2 = F_3$.

Supposons la propriété vraie au rang $k-1$ avec $k \geqslant 2$.

Considérons deux entiers a et b tels que `euclide(a,b)` effectue $k \geqslant 1$ divisions euclidiennes. La première de ces divisions est celle de a par b, c'est à dire $a = bq + r$ avec q et r deux entiers tels que $0 \leqslant r < b$. Si $r = 0$, `euclide(a,b)` ne fait pas de divisions supplémentaires, ainsi $k = 1$ et on est ramené au cas d'initialisation. On prend donc $r > 0$. Comme $\text{pgcd}(a, b) = \text{pgcd}(b, r)$, les $k-1$ divisions restantes sont celles qu'effectue `euclide(b,r)`. Comme $b > r \geqslant 1$, par hypothèse de récurrence au rang $k-1$, on a

$$b \geqslant F_{(k-1)+2} = F_{k+1} \quad \text{et} \quad r \geqslant F_{(k-1)+1} = F_k.$$

Comme $a > b$, q ne peut être égal à 0 car sinon $a = r < b$, ainsi $q \geqslant 1$ et donc

$$a = bq + r \geqslant b + r \geqslant F_{k+1} + F_k = F_{k+2}.$$

La propriété est donc bien vérifiée au rang k, ce qui assure l'hérédité.

Remarque : l'hypothèse $a > b \geqslant 1$ n'est pas restrictive, on peut toujours s'y ramener :
- si $b = 0$, `euclide(a,b)` n'effectue aucune division euclidienne.
- si $b = a$, alors $a = 1 \times b + 0$ donc $\text{pgcd}(a, b) = \text{pgcd}(a, 0) = a$, et donc une division euclidienne suffit.
- si $b > a$, alors $a = 0 \times b + a$ avec $a < b$ donc $\text{pgcd}(a, b) = \text{pgcd}(b, a)$, et on est ramené au cas où $a \geqslant b$.

Corollaire 2 (Théorème de Lamé, 1844) *Soit a et b deux entiers tels que $a > b \geqslant 1$. Le nombre k de divisions eucldiennes effectuées par `euclide(a,b)` est inférieur ou égal à*

$$\frac{\ln b}{\ln \phi} + 1,$$

où $\phi = \dfrac{1+\sqrt{5}}{2}$ est le nombre d'or.

En particulier, le nombre de divisions euclidiennes est inférieur ou égal à cinq fois le nombre de chiffres de b.

Preuve :
On sait déjà que $F_{k+1} \leqslant b$. On veut en déduire une majoration de $k+1$, en «inversant» cette formule. Mais l'expression de F_{k+1} dans la formule de Binet ne le permet pas de façon immédiate. On procède alors ainsi : on montre par récurrence [2] que pour tout entier $n \in \mathbb{N}^*$, $F_n \geqslant \phi^{n-2}$.

On a alors $b \geqslant F_{k+1} \geqslant \phi^{k-1}$, d'où $\ln b \geqslant (k-1)\ln\phi$ par croissance de ln. Enfin comme $\phi > 1$, $\ln\phi > 0$, donc

$$k - 1 \leqslant \frac{\ln b}{\ln \phi}.$$

Si b s'écrit avec N chiffres, alors $b < 10^N$ donc $\ln b < N \ln 10$ et $k - 1 < \dfrac{N \ln 10}{\ln \phi}$. Comme $\dfrac{\ln 10}{\ln \phi} \approx 4.78 < 5$, on obtient $k - 1 < 5N$ donc $k < 5N + 1$ ce qui donne bien $k \leqslant 5N$.

Ce résultat est remarquablement positif :

1. tout d'abord, la majoration du nombre de divisions ne dépend que du plus petit des entiers. Par exemple, que l'on calcule $\text{pgcd}(34, 21)$ ou $\text{pgcd}(3400000000, 21)$, le nombre de division est inférieur à 10 car 21 possède deux chiffres.

2. C'est vrai pour $n = 1$ car $F_1 = 1$ et $\phi^{-1} < 1$ car $\phi > 1$. L'hérédité repose sur l'identité suivante :

$$F_n = F_{n-1} + F_{n-2} \geqslant \phi^{n-3} + \phi^{n-4} = \phi^{n-3}(1 + \frac{1}{\phi}) = \phi^{n-3} \times \phi = \phi^{n-2}.$$

2. dans le pire des cas, le nombre de divisions à effectuer est proportionnel au logarithme de l'entier b. Par exemple, si je multiplie b par $1000 = 10^3$, j'obtiens un nombre avec 3 chiffres de plus, le nombre maximum de divisions nécessaires est alors $5(N+3) = 5N+15$, donc seulement 15 de plus que pour b. C'est en ce sens, que l'algorithme d'Euclide est performant.

Pour finir, la proposition suivante montre que les inégalités de la proposition 1 sont optimales, ces inégalités sont des égalités lorsque a et b sont deux nombres de Fibonacci consécutifs. On dit que c'est le *pire des cas*.

Proposition 3 *Soit $n \in \mathbb{N}^*$. Pour calculer le pgcd de F_{n+2} et F_{n+1}, l'algorithme* `euclide` *effectue exactement n divisions euclidiennes. De plus, leur pgcd vaut* 1.

Preuve :
Traitons d'abord le cas particulier $n = 1$. Comme $F_3 = 2$ et $F_2 = 1$, il suffit de calculer $\text{pgcd}(2,1)$ qui ne nécessite qu'une seule division celle de 2 par 1, car le premier reste obtenu vaut 0.

Prenons désormais $n \geqslant 2$. La clé est la formule de récurrence $F_{n+2} = F_{n+1} + F_n$. Comme la suite (F_n) est strictement croissante à partir du rang 2, on a pour $n \geqslant 2$, $F_n < F_{n+1}$ ce qui prouve que le reste de la division euclidienne de F_{n+2} par F_{n+1} est F_n (attention c'est faux pour $n = 1$, le reste de la division euclidienne de $F_3 = 2$ par $F_2 = 1$ vaut 0 et donc pas $F_1 = 1$). On en déduit que

$$\begin{aligned}
\text{pgcd}(F_{n+2}, F_{n+1}) &= \text{pgcd}(F_{n+1}, F_n) \\
&= \text{pgcd}(F_n, F_{n-1}) \\
&= \cdots \\
&= \text{pgcd}(F_3, F_2) = \text{pgcd}(2,1) \\
&= \text{pgcd}(1,0) = 1.
\end{aligned}$$

Chaque ligne d'égalité de pgcd correspond à une division euclidienne effectuée par `euclide`, il y en a bien n au total.

Par exemple, pour $n = 7$, on a $F_8 = 21$ et $F_9 = 34$. Alors `euclide(34,21)` effectue 7 divisions euclidiennes. Signalons par exemple que `euclide(3400000000,21)` n'en effectue que 4.

Il est facile de modifier l'algorithme d'Euclide de façon qu'il renvoie en plus du pgcd, le nombre de divisions euclidiennes nécessaires à son exécution. Il suffit d'insérer une variable `compteur` initialisée à 0 et qui à chaque itération est incrémentée de 1 par l'affectation `compteur` $\leftarrow$ `compteur` $+ 1$. On renvoie à la fin en plus du pgcd, la valeur de `compteur`. Voici une version Maple :

```
> euclidebis:=proc(a::nonnegint,b::nonnegint)
>    local u,v,r, compteur;
>    u:=a;
>    v:=b;
>    compteur:=0;
>    while v<>0 do
>       r:=irem(u,v);
>       u:=v;
>       v:=r;
>       compteur:=compteur+1;
>    od;
>    RETURN(u,compteur);
> end:
```

3 Un point faible de Fibonacci : la récursivité

Dans cette section, nous allons présenter une méthode importante de programmation appelée *récursivité* qui réserve bien des surprises lorsqu'on l'utilise pour calculer les nombres de Fibonacci.

3.1 Présentation et premier exemple

La récursivité pourrait avoir la morale suivante : on résout quelque chose de compliqué en fonction d'une version un peu moins compliquée de la même chose jusqu'à arriver à quelque chose de très facile, puis on revient en arrière pour résoudre le problème initial.

Un algorithme récursif fait appel à lui-même dans le corps de sa propre définition, il est constitué de deux éléments remarquables :
- **le cas de base** (correspond à la version facile du problème)
- **la formule de propagation ou d'hérédité** (permet de passer à une version plus simple)

Par exemple l'algorithme d'Euclide pour calculer le pgcd de deux entiers peut s'écrire de manière récursive :
- le cas de base est lorsque $b = 0$, dans ce cas $\text{pgcd}(a, 0) = a$.
- la formule de propagation est $\text{pgcd}(a, b) = \text{pgcd}(b, r)$ où r est le reste de la division euclidienne de a par b. En effet, le problème «trouver $\text{pgcd}(b, r)$» est bien «une version plus simple» du problème «trouver $\text{pgcd}(a, b)$» puisque $0 \leqslant r < b$.

Nous appelons `euclideрec` notre algorithme d'Euclide écrit en récursif.

algorithme : `euclideрec`
Entrées : $a, b \in \mathbb{N}$
Résultat : $\text{pgcd}(a, b)$
  Si $b = 0$ alors
    Renvoyer a
  Sinon
    Renvoyer `euclideрec(b,r)`
où r est le reste de la divison de a par b.

```
> euclideрec:=proc(a::nonnegint,b::nonnegint)
>    if b=0 then
>       RETURN(a);
>    else
>       RETURN(euclideрec(b,irem(a,b)));
>    fi;
> end:
```

Examinons la trace de `euclideрec(18459,3809)` que nous livre Maple :

```
> trace(euclideрec);
> euclideрec(18459,3809);
enter euclideрec, args = 18459, 3809
enter euclideрec, args = 3809, 3223
enter euclideрec, args = 3223, 586
enter euclideрec, args = 586, 293
enter euclideрec, args = 293, 0
exit euclideрec (now in euclideрec) = 293\}
exit euclideрec (now in euclideрec) = 293\}
exit euclideрec (now in euclideрec) = 293\}
exit euclideрec (now in euclideрec) = 293\}
exit euclideрec (now at top level) = 293\}
```

293

La fonction `euclideрec` est appelée une première fois avec les arguments 18459 et 3809.

Ensuite, par la formule de propagation elle appelle `euclideрec` avec les arguments 3809 et 3223. La fonction `euclideрec` est ainsi successivement appelée avec les arguments 3223 et 586, 3223 et 586, 586 et 293 et enfin 293 et 0. C'est le cas de base et donc le dernier appel.

On aperçoit ici un des avantages de la programmation récursive : le code est plus simple à écrire mais aussi à lire. Nous n'avons pas eu à recourir à l'affectation et donc aux variables, contrairement à la version itérative. De plus dans notre exemple, `euclideрec` effectue le même nombre de divisions que `euclide`. Les deux algorithmes ont la même complexité. On va voir que ce n'est pas toujours le cas.

3.2 Le cas de la suite de Fibonacci

La suite de Fibonacci (F_n) est un cas particulier de suite récurrente. On peut écrire un programme récursif nommé `fiborec` qui renvoie la valeur du n-ième terme F_n. En effet :

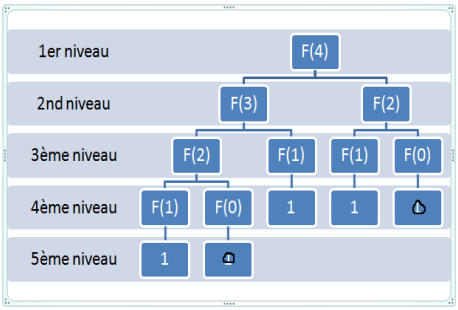
– les conditions initiales $F_0 = 0$ et $F_1 = 1$ constituent le cas de base
– la formule de récurrence $F_n = F_{n-1} + F_{n-2}$ constitue la formule de propagation.

algorithme : `fiborec`
Entrées : $n \in \mathbb{N}$
Résultat : F_n
  Si $n = 0$ ou $n = 1$ alors
    Renvoyer n
  Sinon
    Renvoyer `fiborec(n-1)+fiborec(n-2)`

```
> fiborec:=proc(n::nonnegint)
>    if n=0 or n=1 then
>       RETURN(n);
>    else
>       RETURN(fiborec(n-1)+fiborec(n-2));
>    fi;
> end:
```

Cet algorithme est inconstestablement plus facile à écrire que l'algorithme itératif `fibo` de la première section.

Testons cet algorithme avec $n = 5, 10, 15, 20, 25$. Essayons maintenant avec 30 puis 40. Il faut l'avouer, ça a l'air de prendre beaucoup de temps. Ce n'est vraiment pas efficace. Essayons de comprendre ce qui se passe et donc ce que calcule l'ordinateur. Pour cela, voici un arbre qui représente les étapes des calculs exécutés pour `fiborec(4)`.



On voit que F_2 a été calculé deux fois, on a fait 4 additions et on a appelé 9 fois la fonction `fiborec`.

Nos remarques sont confirmées par la propre fonction trace de Maple :

```
> trace(fiborec);
> fiborec(4);
 enter fiborec, args = 4
 enter fiborec, args = 3
 enter fiborec, args = 2
enter fiborec, args = 1
 exit fiborec (now in fibo) = 1\}
 enter fiborec, args = 0
 exit fiborec (now in fiborec) = 0\}
 exit fiborec (now in fiborec) = 1\}
 enter fiborec, args = 1
 exit fiborec (now in fiborec) = 1\}
 exit fiborec (now in fiborec) = 2\}
 enter fiborec, args = 2
 enter fiborec, args = 1
 exit fiborec (now in fiborec) = 1\}
 enter fiborec, args = 0
 exit fiborec (now in fiborec) = 0\}
 exit fiborec (now in fiborec) = 1\}
 exit fiborec (now at top level) = 3\}
```

3

Si on trace un arbre pour `fiborec(5)`, on voit que l'on fait 7 additions et que la fonction `fiborec` a été appelée 15 fois. Beaucoup de calculs sont inutilement répétés. L'algorithme évalue deux fois `fiborec(3)` et trois fois `fiborec(2)`. Ce processus se termine lors de l'évaluation de `fiborec(0)` ou de `fiborec(1)` qui constituent le cas de base.

Alors faut-il jeter à la poubelle `fiborec` ? Pas forcément. On peut considérablement l'améliorer en lui demandant de garder en mémoire les calculs intermédiaires. Ceci est possible avec Maple avec l'instruction `option remember ;` que l'on insère à la deuxième ligne du programme. Cette fois ci, ça marche mieux et semble aussi rapide qu'avec `fibo`. On ne peut toutefois oublier le défaut majeur de notre programme, il nécessite beaucoup de mémoire, sa «complexité spatiale» est mauvaise.

3.3 Comparaison de la complexité des algorithmes récursif et itératif `fiborec` et `fibo`

1. Pour calculer F_n avec l'algorithme itératif `fibo`, on effectue $n-1$ additions. On ne parlera pas des affectations que nous négligerons (2 initiales et 3 affectations pour chaque itération donc au total $2+3(n-1)$). On négligera aussi les comparaisons que l'on a à faire à chaque itération pour savoir si la variable k est comprise entre 2 et n.
   On dit que la complexité «temporelle» de `fibo` est linéaire car en $O(n)$.
2. Pour l'algorithme récursif `fiborec` c'est plus compliqué.
   Notons a_n le nombre d'appels à la fonction `fiborec` et s_n le nombre d'additions nécessaires pour calculer F_n. On a alors
   $$a_0 = a_1 = 1 \quad \text{et} \quad \forall n \geqslant 2,\ a_n = 1 + a_{n-1} + a_{n-2}.$$
   En effet, l'appel initial de `fiborec(n)` entraîne l'appel de `fiborec(n-1)` et de `fiborec(n-2)`. De même
   $$s_0 = s_1 = 0 \quad \text{et} \quad \forall n \geqslant 2,\ s_n = 1 + s_{n-1} + s_{n-2}.$$
   En effet comme `fiborec(n)=fiborec(n-1)+fiborec(n-2)`, une addition est nécessaire plus celles pour calculer `fiborec(n-1)` et `fiborec(n-2)`.
   On voit donc en particulier, que pour $n \geqslant 2$, $s_n > s_{n-1}+s_{n-2}$ et comme $s_2 = F_2 = 1$ et $s_3 = F_3 = 3$, on a pour $n \geqslant 2, s_n > F_n$ et donc s_n devient énorme lorsque n grandit. En effet, on a déjà vu dans la première section que F_n était équivalent au voisinage de $+\infty$ à $\dfrac{\phi^n}{\sqrt{5}}$. Aucun ordinateur, aussi rapide soit-il, ne peut par cet algorithme récursif calculer F_{50} qui vaut 12586269025.
   On dit que la complexité «temporelle» de `fiborec` est exponentielle car en $O(\phi^n)$ avec $\phi > 1$.
   Voici un tableau indiquant le temps de calcul nécessaire à un ordinateur pour exécuter des algorithmes de complexités différentes. On suppose que notre ordinateur exécute une opération élémentaire en 100 nanosecondes (10^{-7} s). On lit en abscisse la complexité de l'algorithme et en ordonnée la taille de l'entrée.

	$\log_2 n$	n	$n.\log_2 n$	n^2	n^3	2^n
10^2	0.66 μ s	10 μ s	66 μ s	1 ms	0.1s	4×10^{15} ans
10^3	1 μ s	100 μ s	1 ms	0.1s	100 s	
10^4	1.3 μ s	1 ms	13 ms	10 s	1 jour	
10^5	1.6 μ s	10 ms	0.1 ms	16 min	3 ans	
10^6	2 μ s	100 ms	2 s	1 jour	3100 ans	
10^7	2.3 μ s	1 s	23 s	115 jours	3 ans	

Pour finir, remarquons que la fonction `rsolve` de Maple nous donne une formule explicite pour les suites (a_n) et (s_n).

```
> rsolve({a(n)=1+a(n-1)+a(n-2),a(0)=1,a(1)=1},a);
```

$$-\frac{4}{5}\,\frac{\sqrt{5}\,(-\frac{2}{-\sqrt{5}+1})^n}{-\sqrt{5}+1} + \frac{4}{5}\,\frac{\sqrt{5}\,(-\frac{2}{\sqrt{5}+1})^n}{\sqrt{5}+1} - 1$$

```
> rsolve({s(n)=1+s(n-1)+s(n-2),s(0)=0,s(1)=0},s);
```

$$-1 + \frac{2}{5}\,\frac{\sqrt{5}\,(\frac{2}{\sqrt{5}-1})^n}{\sqrt{5}-1} + \frac{2}{5}\,\frac{\sqrt{5}\,(-\frac{2}{\sqrt{5}+1})^n}{\sqrt{5}+1}$$

4 Plus rapide que les lapins ... de Fibonacci ?

Dans cette section, nous proposons une dernière méthode permettant de calculer les nombres de Fibonacci. Elle repose sur l'écriture matricielle de la relation de récurrence $F_{n+2} = F_{n+1} + F_n$. En effet si l'on pose

$$X_n = \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} \quad \text{et} \quad A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix},$$

un calcul immédiat montre que $X_{n+1} = AX_n$. La suite de matrices $(X_n)_n$ est donc une «suite géométrique de raison A» et de premier terme X_0. Par récurrence immédiate, on a pour tout $n \in \mathbb{N}, X_n = A^n X_0$.

De plus si $A^n = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$, alors $A^n X_0 = A^n \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} b \\ d \end{pmatrix} = \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix}$.

Le nombre de Fibonacci F_n est donc le coefficient de la première ligne et de la deuxième colonne de la matrice A^n. Calculer F_n revient donc à calculer le produit matriciel A^n.

Pour cela nous allons présenter la méthode importante d'exponentiation rapide qui permet de calculer efficacement les puissances n-ièmes d'un entier a. Il nous suffira alors de l'adapter au produit matriciel.

4.1 La méthode d'exponentiation rapide

La méthode naïve pour calculer a^n revient à faire $a \times \cdots \times a$ donc à effectuer $n-1$ multiplications.

algorithme : `exponaïf`
Entrées : $a \in \mathbb{N}^*,\ n \in \mathbb{N}^*$
Résultat : a^n
Variables : P, k
$P \leftarrow a$
Pour k de 1 à $n-1$, faire
  $P \leftarrow P \times a$
Fin du pour
Renvoyer P

Il est possible de faire beaucoup moins de multiplications, en écrivant n en base 2. Examinons les exemples suivants :

- on écrit $a^8 = \left((a^2)^2\right)^2$. Trois élévations au carré sont alors nécessaires au lieu de 7 multiplications dans la méthode naïve. Plus généralement, pour calculer a^{2^p} par cette méthode, seulement p multiplications sont nécessaires au lieu de $2^p - 1$.
- on a $a^{11} = a \times a^{10} = a \times (a^2)^5 = a \times (a^2) \times (a^2)^4 = a \times (a^2) \times \left((a^2)^2\right)^2$. Il y a 2 multiplications et 3 élévations au carré (a^2 n'est calculé qu'une fois) soit au total 5 multiplications au lieu de 10.

Cette méthode porte le nom d'exponentiation rapide, elle repose donc sur le principe suivant :

$$\begin{cases} a^0 &= 1 & \\ a^n &= \left(a^2\right)^{\frac{n}{2}} & \text{, si } n \text{ est pair et } n > 0 \\ a^n &= a \times (a^2)^{\frac{n-1}{2}} & \text{, si } n \text{ est impair} \end{cases}$$

Elle se prête bien à une programmation récursive.

En effet $n = 0$ est le cas de base, les deux dernières lignes constituent la formule de propagation.

algorithme : `exporap`
Entrées : $a \in \mathbb{N}^*,\ n \in \mathbb{N}$
Résultat : a^n
Si $n = 0$ alors
  renvoyer 1
Sinon
  si n est impair alors
    renvoyer $a \times$ `exporap`$(a^2, (n-1)/2)$
  sinon
    renvoyer `exporap`$(a^2, n/2)$
  Fin du si
Fin du si ;

```
> exporap:=proc(a::posint,n::nonnegint)
>    if n=0 then
>       RETURN(1);
>    elif irem(n,2)=1 then
>       RETURN(a*exporap(a^2,(n-1)/2));
>    else  RETURN(exporap(a^2,n/2));
>    fi;
> end;
```

Déterminons la complexité de `exporap`. Pour cela, examinons auparavant la trace de `exporap(3,11)` qui renvoie 177147.

```
enter exporap, args = 3, 11
```

```
 enter exporap, args = 9, 5

 enter exporap, args = 81, 2

 enter exporap, args = 6561, 1

 enter exporap, args = 43046721, 0

exit exporap (now in exporap) = 1\}

 exit exporap (now in exporap) = 6561\}

exit exporap (now in exporap) = 6561\}

 exit exporap (now in exporap) = 59049\}

exit exporap (now at top level) = 177147\}
```

La fonction `exporap` a été appelée cinq fois. Lors de chaque appel,
- l'exposant, c'est-à-dire le deuxième argument (initialement à 11) est divisé au moins par 2 pour tomber à 0 $(11, 5, 2, 1, 0)$.
- le premier argument, c'est à dire le nombre que l'on exponentie est élevé au carré $(3, 9, 81, 6561, 43046721)$.

Passons au cas général, nous allons prouver

Proposition 4 *Soit $a \in \mathbb{N}^*$ et $n \in \mathbb{N}$. Pour calculer a^n, l'algorithme récursif* ***`exporap`*** *effectue au maximum*

$$2(\log_2(n) + 1) \quad \textit{multiplications.}$$

Preuve :
Soit I le nombre de fois où la fonction `exporap` est appelée pour calculer `exporap(a,n)`. Pour $k \in \{0, \ldots, I\}$, on note u_k la valeur de l'exposant après k appels récursifs. On a $u_0 = n$. Remarquons aussi que lors du dernier appel, l'exposant vaut 0 (cas de base) donc $u_I = 0$ et pour l'avant dernier appel, l'exposant vaut 1 donc $u_{I-1} = 1$.

De plus à chaque appel, l'exposant est au moins divisé par 2, donc $u_k \leqslant \frac{u_{k-1}}{2}$ et ainsi par récurrence immédiate

$$u_{I-1} \leqslant \frac{u_0}{2^{I-1}} = \frac{n}{2^{I-1}}.$$

On a donc $1 \leqslant \dfrac{n}{2^{I-1}}$ d'où en prenant le log en base 2, on obtient

$0 \leqslant \log_2(n) - (I-1)$ donc $I \leqslant \log_2(n) + 1$.

Enfin, pour chaque appel de `exporap`, si l'exposant est pair, il y a une élévation au carré, et s'il est impair, il y a une multiplication plus une élévation au carré, donc dans les deux cas, deux multiplications au maximum sont effectuées à chaque appel. Comme le nombre d'appel est inférieur ou égal à $\log_2(n)+1$, le résultat est démontré.

Remarques :
- On dit que la complexité de `exporap` est logarithmique, c'est incroyablement mieux que `exponaïf` dont la complexité en $O(n)$ est dite linéaire.
- On peut démontrer que le pire des cas est atteint lorsque l'on choisit pour exposant n un nombre de Mersenne, c'est à dire un entier de la forme $2^p - 1$.

Pour finir, donnons une version itérative `exporapiter` de l'algorithme d'exponentiation rapide. Elle a la même complexité que la version récursive.

algorithme : exporapiter
Entrées : $a \in \mathbb{N}^*$, $n \in \mathbb{N}$
Résultat : a^n
Variables : exposant,itere,produit
  exposant $\leftarrow n$
  itere $\leftarrow a$
  produit $\leftarrow 1$
  Tant que exposant est non nul
    Si n est impair
      produit $\leftarrow$ itere $\times$ produit
      exposant$\leftarrow$ exposant -1
    Sinon
      itere $\leftarrow$ itere $\times$ itere
      exposant $\leftarrow$ exposant/2
    Fin du Si
  Fin du Tant que
  Renvoyer produit

```
> exporapiter:=proc(a::posint,n::nonnegint)
>    local exposant,itere,produit;
>    exposant:=n;
>    itere:=a;
>    produit:=1;
>    while exposant<>0 do
>       if irem(exposant,2)=1 then
>          produit:=itere*produit;
>          exposant:=exposant-1;
>       else
>          itere:=(itere)^2;
>          exposant:=exposant/2;
>        fi;
>    od;
>    RETURN(produit);
> end:
```

4.2 Application au calcul des nombres de Fibonacci

L'algorithme `exporap` doit être modifié pour fonctionner avec les matrices. En effet, le cas de base n'est plus $a^0 = 1$ mais $A^0 = I_2$ où I_2 désigne la matrice unité de $M_2(\mathbb{R})$. De plus, le produit matriciel en Maple ne se note pas `*` mais `&*` et il faut rajouter la fonction `evalm` qui évalue le résultat du produit. Voici notre nouvelle version appelée `exporapm`.

```
> exporapm:=proc(a,n::nonnegint)
>    if n=0 then
>       RETURN(matrix(2,2,[1,0,0,1]);
>    elif
>       irem(n,2)=1 then
>       RETURN(evalm(a &*exporapm(a^2,(n-1)/2)));
>    else  RETURN(exporapm(evalm(a^2),n/2)));
>    fi;
> end;
```

On applique alors `exporapm` à la matrice A.

```
> A:=matrix(2,2,[[0,1],[1,1]]):
> exporapm(A,50);
> exporapm(A,50)[1,2]; # renvoie le coefficient d'indice 1,2
```

$$\begin{bmatrix} 7778742049 & 12586269025 \\ 12586269025 & 20365011074 \end{bmatrix}$$

$$12586269025$$

On retrouve bien que $F_{50} = 12586269025$.

Enfin, on présente tout ceci dans une fonction que l'on nomme `fiborap` :

```
> fiborap:=n->exporapm(A,n)[1,2]:
```

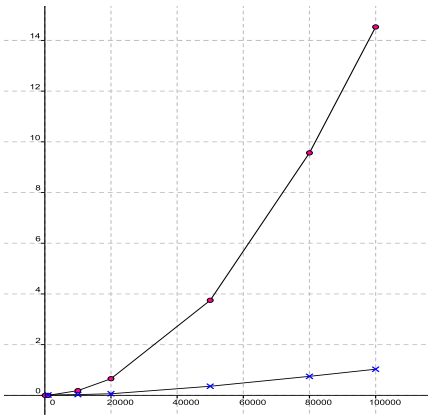
Effectuer le produit de deux matrices 2×2 coûte 8 multiplications et 4 additions d'entiers. Comme, calculer A^n par exponentiation rapide, nécessite au maximun $2(\log_2(n) + 1)$ multiplications matricielles, `fiborap(n)` effectue au maximum $16(\log_2(n) + 1)$ multiplications d'entiers et $8(\log_2(n) + 1)$ additions d'entiers.

Nous disposons à présent de trois algorithmes de calcul des nombres de Fibonacci : `fibo`, `fiborec` et `fiborap`. Observons avec la fonction time de Maple leur efficacité pour calculer F_{50000}.

L'algorithme `fibo` met 3.75 s (secondes) tandis que `fiborap(50000)` est beaucoup plus rapide (on l'espérait) et ne met que 0.359 s. Et pour `fiborec(50000)` ? Et bien Maple est incapable de le calculer, même avec l'option remember, il nous répond «*Error, (in fiborec) too many levels of recursion*»...

Les temps de calcul de `fibo` et `fiborap` sont récapitulés dans le tableau suivant puis représentés graphiquement :

n	100	1000	10000	20000	50000	80000	10^6	10^7
temps mis par `fibo(n)` en s	0	0	0.187	0.656	3.750	9.563	14.531	∞
temps mis par `fiborap(n)` en s	0.016	0.015	0.031	0.063	0.359	0.75	1.031	103



Remarque : nous avons dit que l'algorithme `fibo` avait une complexité linéaire car il effectuait $n-1$ additions pour calculer le nombre F_n. Pourtant la courbe représentant le temps de calcul de `fibo(n)` ne semble pas être une droite. Cela serait le cas si l'opération «additionner deux entiers» prenait un temps constant. Mais les nombres de Fibonacci devenant très grands, les additions à effectuer sont de plus en plus coûteuses.

Pour terminer signalons que l'exponentiation rapide n'est pas un algorithme gadget :

– on peut l'adapter très facilement pour calculer des puissances modulo un entier, on parle alors de puissance modulaire. C'est par exemple utilisé en cryptographie,

– la propre fonction puissance de Maple applique essentiellement cette méthode d'exponentiation rapide. Ce n'est donc pas étonnant que la fonction `fibo4` ci-dessous très simple à programmer calcule F_n avec des performances comparables à `fiborap`.

```
> fibo4:=n->evalm(A^n)[1,2]:
> fibo4(50);
```

12586269025

n	100	1000	10000	20000	50000	80000	10^6	10^7
temps mis par `fibo4(n)` en s	0	0	0.047	0.047	0.234	0.578	0.922	100

Remerciements

Remerciements à Dominique Hoareau pour ses encouragements, à Bruno Harington pour sa relecture et à Sami Bakri pour ses remarques pertinentes.

Références

[1] Cormen, Leiserson, Rivest, Stein, *Algorithmique*, Dunod (2010)
[2] Porcheron Lionel, *MAPLE*, Dunod (2006)
[3] Jean-Pierre Demailly, *Analyse numérique et équations différentielles*, EDP sciences (2006)
[4] Verdier, Bordelles, Schott, Seitz, *Variations eucldiennes*, Repères-IREM n°73 (octobre 2008)
[www] *Calcul mathématique avec Sage*, Licence Creative Commons (2010)