

# Suite de Fibonacci

## Algo & Prog avec R

---

A. Malapert, B. Martin, M. Pelleau, et J.-P. Roy

18 avril 2019

Université Côte d'Azur, CNRS, I3S, France  
`firstname.lastname@univ-cotedazur.fr`

# Suite de Fibonacci

## Définition

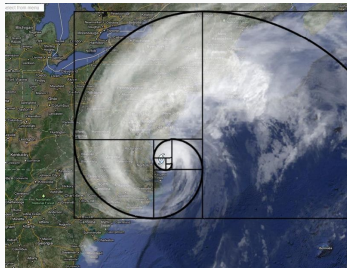
Le  $n$ -ème terme est défini ainsi :

$$\mathcal{F}_n = \mathcal{F}_{n-1} + \mathcal{F}_{n-2}$$

et  $\mathcal{F}_0 = 0$ ,  $\mathcal{F}_1 = 1$ .

## Premiers termes

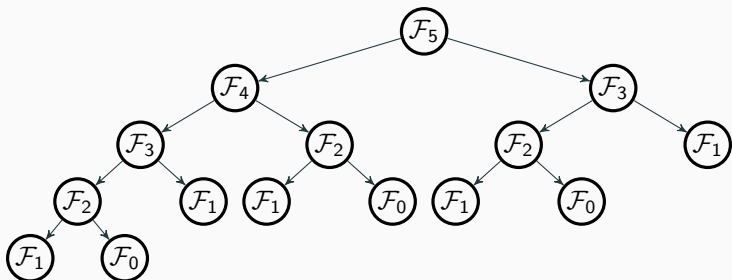
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, ...



D'autres d'images de suites de Fibonacci harmonieuses.

## Récursion simple (top-down)

```
F <- function(n) {  
  if( n < 2) return(n)  
  else return(F(n-1) + F(n-2))  
}
```



**Catastrophe ! La complexité de l'algorithme est exponentielle !**  
Plus de 15 secondes pour calculer  $F(35)$  !

Programmer une fonction qui se souvient des calculs déjà effectués !

## Exemple avec Fibonacci

- ▶ Je calcule  $\mathcal{F}_{35}$  qui demande le calcul de  $\mathcal{F}_{34}$ .
- ▶ Je calcule  $\mathcal{F}_{36}$  qui demande une seule addition si je suis capable de me souvenir de  $\mathcal{F}_{35}$  et de  $\mathcal{F}_{34}$ .

## Comment ?

- ▶ Nous allons gérer un dictionnaire privé à la fonction qui va contenir tous les couples  $n : v$  tels que  $\mathcal{F}_n == v$  ait déjà été calculé !
- ▶ Ici, le dictionnaire est un vecteur tel que  $\mathcal{F}_n$  est à la position  $n + 1$ .
  - ▶ les indices de la suite commencent à 0.
  - ▶ les indices du vecteur commencent à 1.
- ▶ Le dictionnaire joue le rôle de mémoire cache.

# Portée des variables

Jusqu'à présent, dans plusieurs fonctions, nous avons introduit des variables qui n'étaient pas des paramètres de la fonction, souvent un compteur `i` ou un accumulateur `acc`.

- ▶ Une telle variable est dite **locale** à la fonction et n'a rien à voir avec une variable de même nom existant en-dehors de cette fonction !
- ▶ Une variable définie en-dehors de toute fonction est **globale**.

```
> i <- 42
> foo <- function() {print(i); i <- 33; print(i)}
> foo()
[1] 42 # globale
[1] 33 # locale
> i # globale
[1] 42
```

- ▶ Les modifications apportées à une variable globale sont locales !
- ▶ Conclusion : les variables introduites dans une fonction sont locales !
- ▶ Pourquoi R a-t-il fait ce choix ? Pour décourager autant que possible l'utilisation de variables globales ! Dont acte ...

# Modifier quand même une variable globale !

## Opérateur «←»

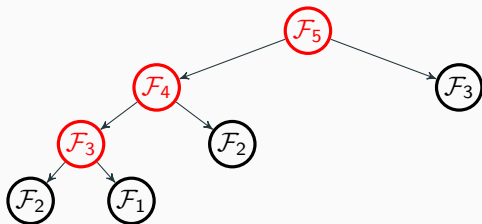
Les modifications apportées à une variable globale sont globales !

```
> i <- 42
> foo <- function() {print(i); i <<- 33; print(i)}
> foo()
[1] 42 # Globale.
[1] 33 # Globale aussi.
> i # Globale toujours !
[1] 33
```

# Mémo-fonction de Fibonacci

```
cache <- c(0, 1, 1)
F <- function(n) {
  if(length(cache) <= n) {
    cache[n + 1] <- F(n-1) + F(n-2)
  }
  return(cache[n + 1])
}
```

```
> F(35) # Immédiat !
[1] 9227465
> F(30)
[1] 832040 # Déjà calculé !
```



**Sauvé ! Les complexités temporelles et spatiales sont linéaires !**  
Le calcul de  $F(35)$  est immédiat.

# Limites de la mémo-fonction de Fibonacci

En mettant de côté les dépassements de capacité,

```
> F(1000)
[1] 4.346656e+208
> F(2000)
[1] Inf
```

La récursivité pose toujours problème !

```
> F(10000)
Erreur : C stack usage 7969716 is too close to the limit
```



# Mémo-fonction : cacher le cache !

## Le cache est public !

Modifions le cache juste après la définition de la mémo-fonction.

```
> cache <- c(5, 13, 34)
> F(3)
[1] 47
```

## Utilisons un constructeur pour la fonction F

Une fonction renvoyant une fonction comme résultat !

```
MakeF <- function() {
  cache <- c(0, 1, 1)
  F <- function(n) {
    if(length(cache) <= n) {
      cache[n + 1] <- F(n-1) + F
        (n-2)
    }
    return(cache[n + 1])
  }
  return(F)
}
```

```
> F <- MakeF()
> cache <- c(5, 13, 34)
> F(3)
[1] 2
```

# Suppression de la récursivité (bottom-up)

Il faut construire une itération calculant les termes par ordre croissant.

## Suppression de la récursivité

```
F <- function(n) {  
  cache <- c(0, 1, 1)  
  if(length(cache) <= n) {  
    for(j in seq(from = length(cache) + 1, to = n + 1)) {  
      cache[j] <- cache[j-1] + cache[j-2]  
    }  
  }  
  return(cache[n + 1])  
}
```

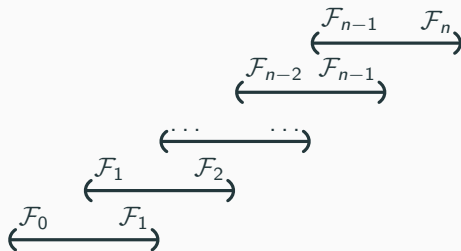
## Plus de problème avec la pile d'appels

```
> F(10000)  
[1] Inf
```

# Réduction de la complexité spatiale

```
F <- function(n) {  
  if(n < 2) return(n)  
  x <- c(0, 1) # F(0), F(1)  
  i <- 2;  
  while(i <= n) {  
    x <- c(x[2], sum(x)) # F(n-1), F(n)  
    i <- i + 1;  
  }  
  return(x[2])  
}
```

La complexité spatiale est maintenant constante !



# Matrice de Fibonacci

```
> library(expm) # pour les puissances de matrice  
> mF <- matrix(c(0, 1, 1, 1), nrow = 2)
```

```
> mF  
      [,1] [,2]  
[1,]    0    1  
[2,]    1    1  
> mF %^% 4  
      [,1] [,2]  
[1,]    2    3  
[2,]    3    5
```

```
> mF %^% 7  
      [,1] [,2]  
[1,]     8   13  
[2,]    13   21  
> mF %^% 10  
      [,1] [,2]  
[1,]    34   55  
[2,]    55   89
```

## Exponentiation rapide

Les méthodes d'exponentiation rapide permettent d'atteindre une complexité logarithmique.

Questions?

Retrouvez ce cours sur le site web

[`www.i3s.unice.fr/~malapert/R`](http://www.i3s.unice.fr/~malapert/R)