Récursivité

Fabrice Lembrez - PSI*

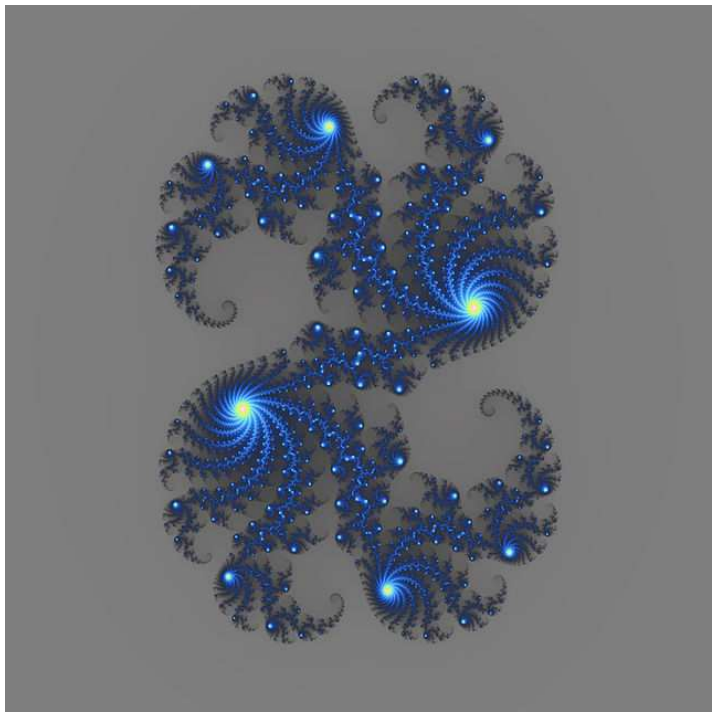
Lycée Pierre de Fermat

Plan

Qu'est-ce que la récursivité ?

La récursivité est très utilisée et théorisée en informatique, mais on la retrouve dans d'autres domaines, à chaque fois qu'on a des démarches auto-référentes.

Exemples en math : définition d'une fonction $\mathcal{C}^k$, suite récurrente, fractales...



Fonctions récursives

Déf - Fonction récursive

Fonction f dont le mode de définition fait appel à f.

Fonctions récursives

Déf - Fonction récursive

Fonction f dont le mode de définition fait appel à f.

Versions récursives de certains calculs

- factorielle : $F(n) = n \times F(n-1)$ et initialisation $F(0) = 1$
- PGCD de a et b :
  - relier PGCD(a,b) et PGCD(b,r) où r=reste de la div de a par b
  - chercher comment initialiser

Fonction récursive de référence : PGCD

Réduction : PGCD(a,b)=PGCD(b,r) où r=reste de la div de a par b
Initialisation : PGCD(a,0)=a.

```
def PGCD(a,b):
    if b==0:
        return a
    else:
        return PGCD(b,a%b)
```

Terminaison :

Fonction récursive de référence : PGCD

Réduction : PGCD(a,b)=PGCD(b,r) où r=reste de la div de a par b
Initialisation : PGCD(a,0)=a.

```
def PGCD(a,b):
    if b==0:
        return a
    else:
        return PGCD(b,a%b)
```

Terminaison : $0 \leq r < |b|$ donc la deuxième valeur diminue strictement

Plan

Pensée récursive, pensée impérative

Pensée impérative

Piloter tout un parcours

Structure du parcours

Initialisation

Pensée récursive

Ramener à un cas plus simple

Cas général : appel

Cas terminal : initialisation

Exemple : dans une version impérative du PGCD, on pourrait écrire toutes les étapes de l'algorithme d'Euclide "tant que le reste n'est pas nul". On est content d'avoir trouvé plus simple.

Pensée récursive, pensée impérative

Pensée impérative	**Pensée récursive**
Piloter tout un parcours	Ramener à un cas plus simple
Structure du parcours	Cas général : appel
Initialisation	Cas terminal : initialisation

Exemple : dans une version impérative du PGCD, on pourrait écrire toutes les étapes de l'algorithme d'Euclide "tant que le reste n'est pas nul". On est content d'avoir trouvé plus simple.

Penser récursif = comme une récurrence

On se place dans l'idée qu'on sait faire les "cas plus simples".

Donc on a juste 2 questions : comment s'y ramener ? Comment cela se termine (cas terminal) ?

Plan

Du récursif mal organisé

Mal organiser les choses pose tout de suite un problème de la terminaison. Ainsi il est très facile de faire des boucles infinies

```
def boucle():
    boucle()
```

Cela provoque une erreur : la machine refuse d'aller au delà de 1000 appels imbriqués environ ("encombrement de la pile des appels").

Du récursif mal organisé

On a également l'exemple de la suite de Syracuse, qui se programme très facilement en récursif

```
def Syracuse(n):
    if n%2==0 :
        return Syracuse(n//2)
    elif n>1 :
        return Syracuse(3*n+1)
```

Mais dont on ne sait pas si elle termine. En effet, l'évolution de la quantité manipulée n'est pas simple du tout !

Organisation correcte et preuve algorithmique

Bonne organisation = preuve d'algorithme

- déterminer une quantité qui décroît strictement à chaque appel
- ce qui assure la pertinence du **"cas terminal"** (ou des cas terminaux)

Et si c'est demandé, la correction (l'algorithme renvoie-t-il ce qu'il faut) se prouve alors en faisant une récurrence sur la quantité en question ; c'est souvent direct.

Plan

Complexité temporelle

Complexité (temporelle)

Cas simple : chaque fonction a un coût fixé C (ex : C=O(1)).

Alors la complexité temporelle est en O(C $\times$ nombre total d'appels).

Cas plus complexes : mélange impératif - récursif.

La complexité s'obtient par une relation de récurrence. On verra des exemples avec les tris.

Complexité spatiale

- à chaque appel de fonction : création d'un "espace de noms" avec ses variables locales
- à chaque retour : libération de ces variables (en Python par le "garbage collector")

Complexité spatiale

- à chaque appel de fonction : création d'un "espace de noms" avec ses variables locales
- à chaque retour : libération de ces variables (en Python par le "garbage collector")

Complexité spatiale

En programmation récursive, la complexité spatiale est la somme des espaces mémoires utilisés par les différentes fonctions ouvertes **simultanément.** On l'obtient en considérant le contenu de la pile des appels, au moment d'encombrement maximum.
Pour un exemple de calcul : voir le tri fusion.

Explosions récursives

Les erreurs suivantes sont archi-calamiteuses : elles font passer d'une complexité linéaire à une complexité exponentielle sans raison.

Deux grosses bourdes en récursif

- appeler deux fois le même calcul quand on peut l'éviter
  Remède : mémoriser le résultat de l'appel dans une variable
- créer des sous listes par slicing s[...]
  Remède : passer la référence de s, des indices de début et de fin

Plan

La suite de Fibonacci

Elle est donnée par

$$F_0 = 0, F_1 = 1, \qquad F_{n+2} = F_n + F_{n+1}$$

On demande de calculer F_n pour un n donné, en temps raisonnable.

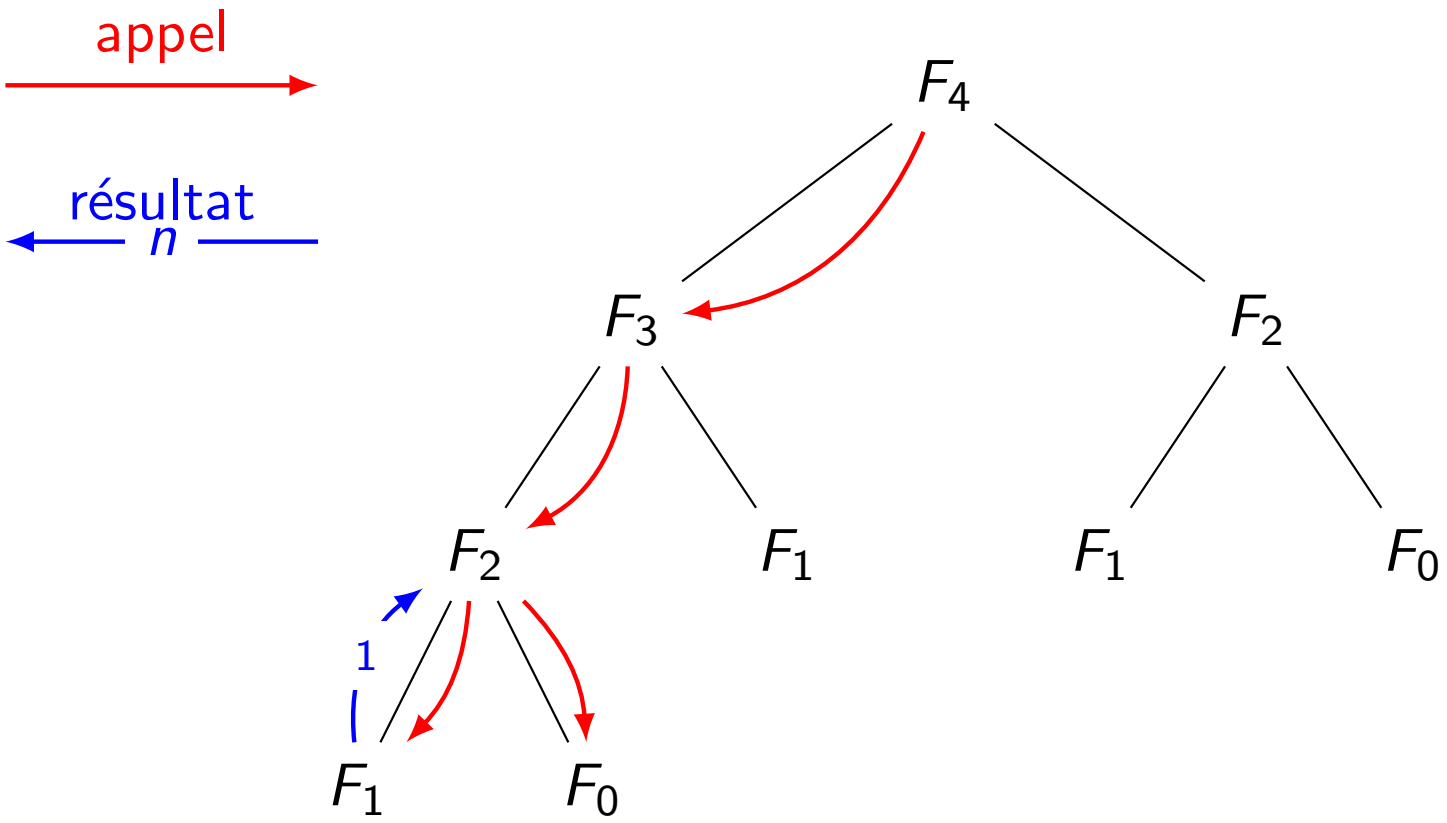
La solution naïve consiste à imiter strictement le mode de définition.

Fibonacci naïf

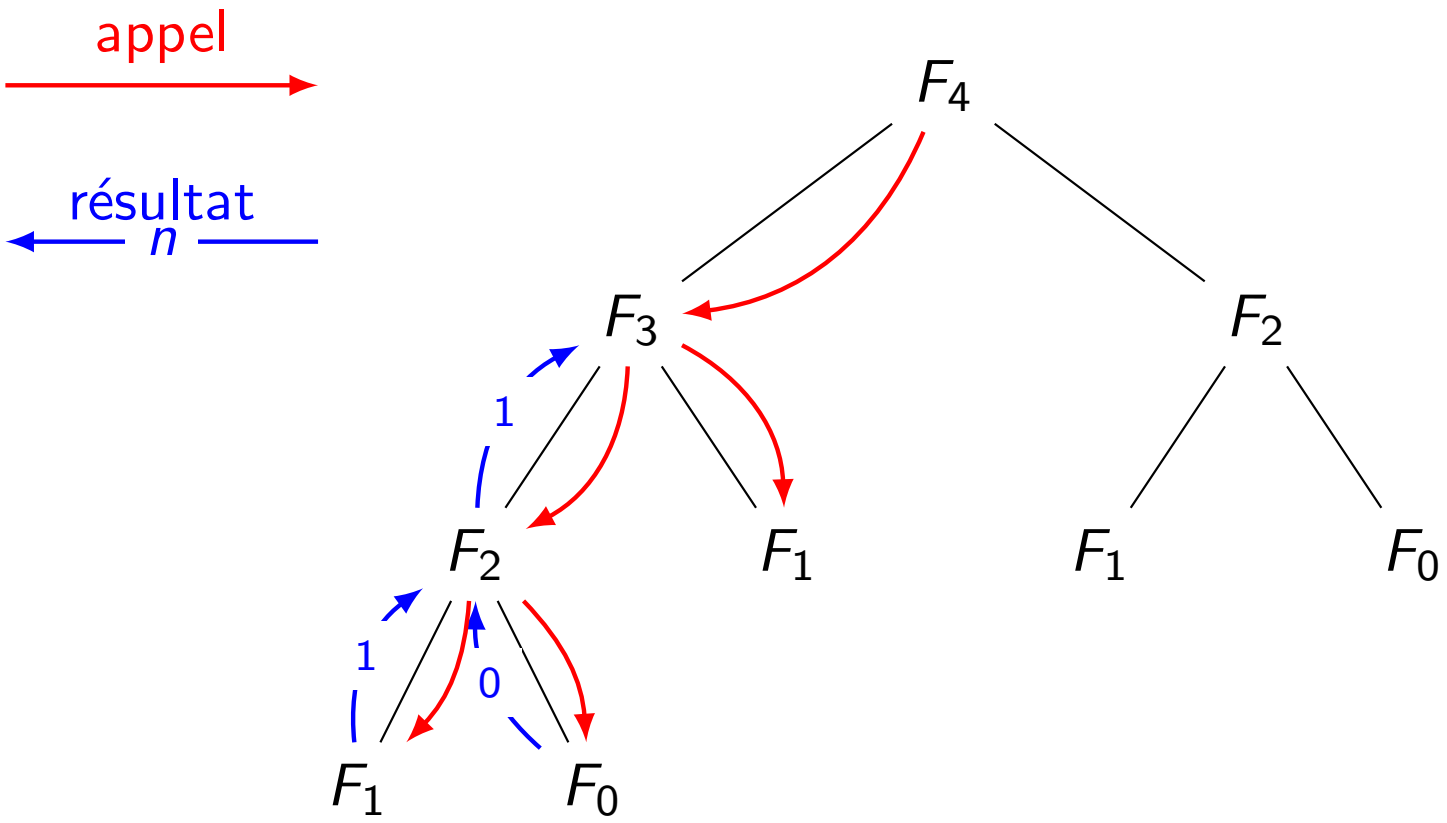
```
def Fibo(n):
    if n<=1
        return n
    else :
        return Fibo(n-1)+Fibo(n-2)
```

Ca coince avant d'attendre n=30... Pourquoi ?

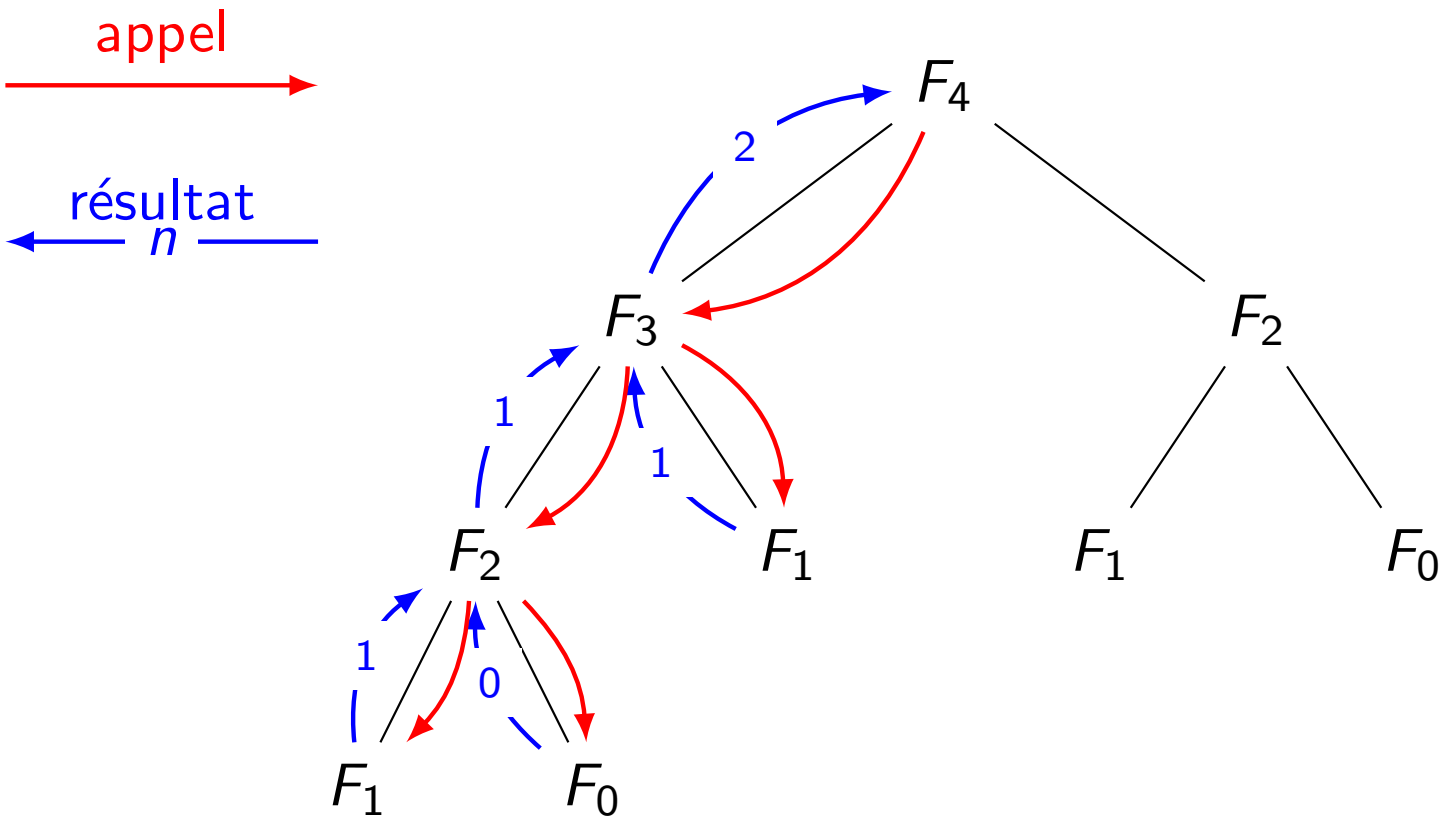
Fibonacci naïf



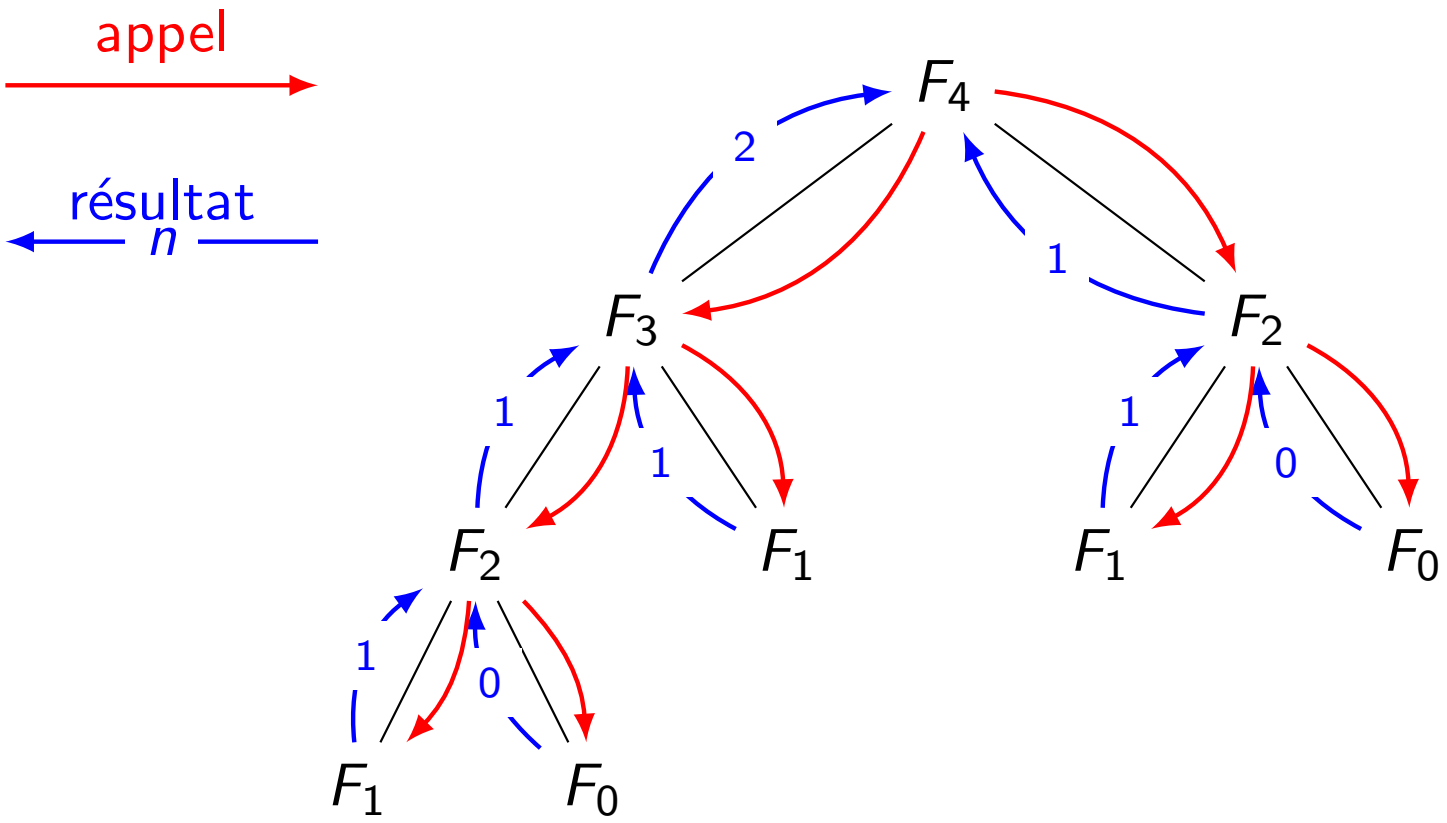
Fibonacci naïf



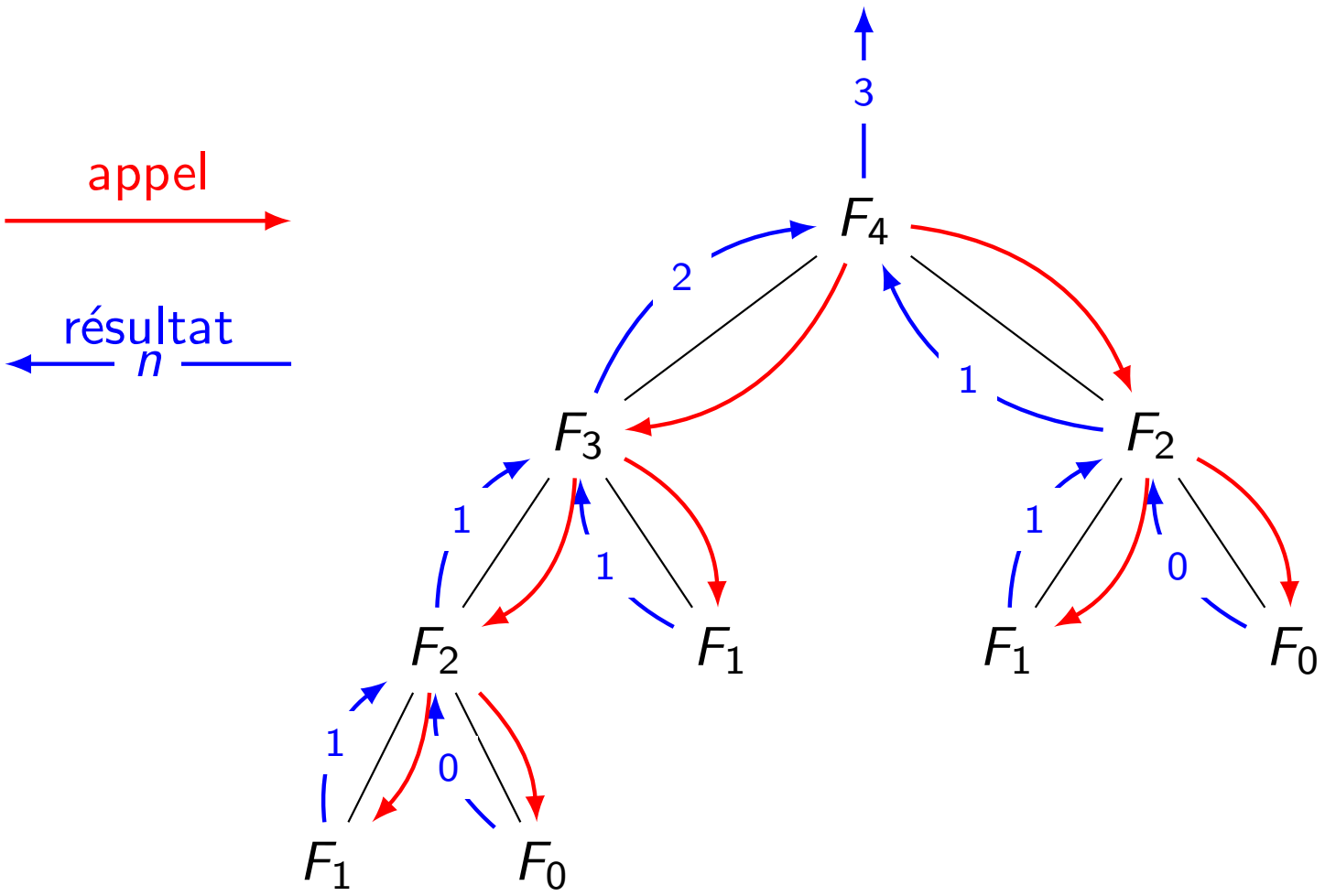
Fibonacci naïf



Fibonacci naïf



Fibonacci naïf



Profondeur de récursion ? Pas plus de n+1, donc ça passe.

Temps de calcul (complexité) : il est en 2^n, c'est ça qui bloque.

Comment faire pour Fibonacci alors ?

Quelques solutions possibles :

- éviter la récursivité !
- stocker les quantités Fibo(n) déjà appelées

  Cette "mémoïsation" peut être faite à la main

  Mais certains langages le font automatiquement

  On la verra ultérieurement
- programmer une version vectorielle via $X_n = \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix}$...

Comment faire pour Fibonacci alors ?

Quelques solutions possibles :

- éviter la récursivité !
- stocker les quantités Fibo(n) déjà appelées

  Cette "mémoïsation" peut être faite à la main

  Mais certains langages le font automatiquement

  On la verra ultérieurement
- programmer une version vectorielle via $X_n = \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix}$...

Il y a aussi une formule explicite

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

Pourquoi ne pas l'employer ?

Plan

Recherche par dichotomie dans une liste triée

On cherche un entier x dans une liste s formée d'entiers et triée en ordre croissant.
- si l'élément ne s'y trouve pas réponse attendue `None`
- s'il y est renvoyer l'indice où il se trouve

Gestion de la zone de recherche [a,b[

- Invariant :

Recherche par dichotomie dans une liste triée

On cherche un entier x dans une liste s formée d'entiers et triée en ordre croissant.
- si l'élément ne s'y trouve pas réponse attendue `None`
- s'il y est renvoyer l'indice où il se trouve

Gestion de la zone de recherche [a,b[

- Invariant : "x est dans s dans la zone d'indices [a,b[, ou bien est absent"
- Essentiellement la taille est divisée par deux à chaque étape
  d'où une complexité temporelle majorée par $\log_2 n$.
- De façon précise s'assurer que la taille de s diminue strictement, en perdant au moins le point m.

Version impérative

```
def recherche(x,s):
    a,b=0,len(s)
    while b>a:
        m=(a+b)//2
        if s[m]>x:
            b=m
        elif s[m]<x:
            a=m+1
        else:
            return m
```

Version récursive

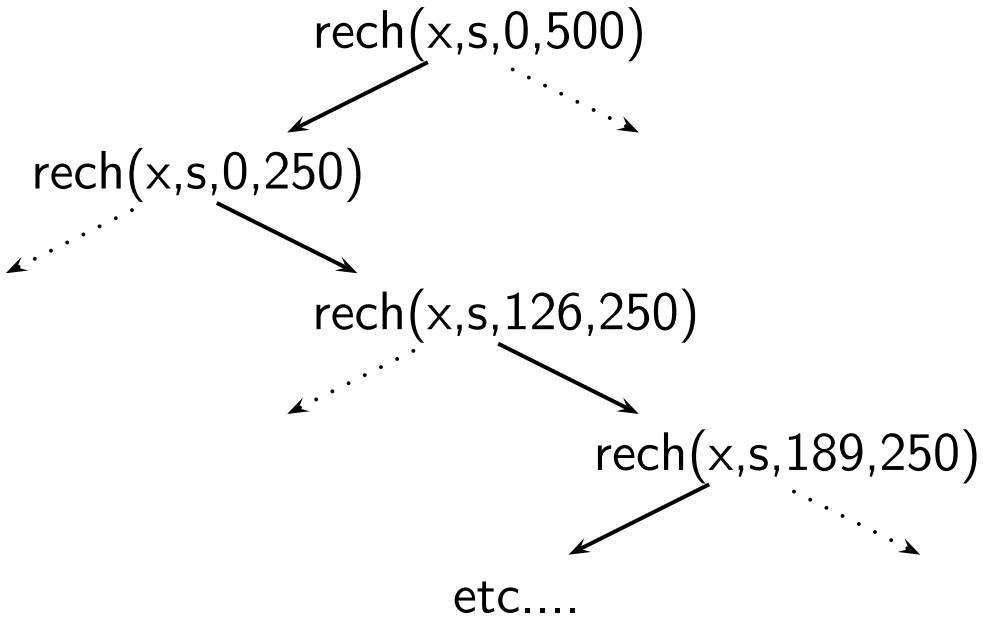
Pour éviter de faire du slicing, on prévoit de faire passer s, et un indice de début et de fin de la zone de recherche.
Mais alors il faut une procédure `rech(x,s,a,b)` et aussi une procédure chargée de mettre en route la récursivité

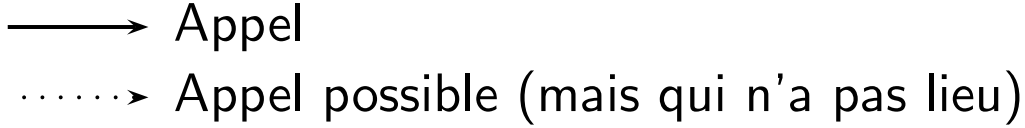
```
def rech(x,s,a,b):
    if [---cas terminal zone vide---]:
        return None
    else: #cas général
        m=(a+b)//2
        [---à écrire---]

def recherche(x,s):
    return rech(x,s,0,len(s))
```

Version récursive

```
def rech(x,s,a,b):
    if b>a:        #cas terminal b<=a implicite : None
        m=(a+b)//2
        if s[m]==x:# sortie brusquée
            return m
        elif s[m] > x:
            return rech(x,s,a,m)
        else:
            return rech(x,s,m+1,b)

def recherche(x,s):
    return rech(x,s,0,len(s))
```

Parcours effectué





Plan

Exponentiation rapide

Principe : comment calculer aussi rapidement que possible a^{128} ? a^{213} ?

Exponentiation rapide

Principe : comment calculer aussi rapidement que possible a^{128} ? a^{213} ?

On peut arriver à des solutions impératives, mais la récursivité rend la réponse particulièrement simple

$$\text{si } n = 2q + r, r \in \{0, 1\}, \qquad a^n = (a^2)^q . a^r$$

Exponentiation rapide

Principe : comment calculer aussi rapidement que possible a^{128} ? a^{213} ?

On peut arriver à des solutions impératives, mais la récursivité rend la réponse particulièrement simple

$$\text{si } n = 2q + r, r \in \{0, 1\}, \qquad a^n = (a^2)^q . a^r$$

Pour la puissance n on se ramène à une puissance q.

Exponentiation rapide

```
def puiss(a,n):
    if n==0:
        return 1
    elif n%2==1:
        return a*puiss(a**2,n//2)
    else:
        return puiss(a**2,n//2)}
```

- Terminaison : décroissance stricte de n.
- Précisément d'une étape à l'autre $n' \leq \frac{1}{2}n$,
  ce qui donne la complexité en $O(\log_2 n)$
- Correction par récurrence forte :
  P(k) "pour $0 \leq k \leq n$, puiss(a,k) renvoie a^k"

Application à la suite de Fibonacci

Peut-on utiliser l'exponentiation rapide pour la suite de Fibonacci ?

Application à la suite de Fibonacci

Peut-on utiliser l'exponentiation rapide pour la suite de Fibonacci ? Oui si on met le problème sous la forme vectorielle $X_{n+1} = AX_n$

$$X_n = \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix}, \qquad A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \qquad X_n A^n X_0$$

Application à la suite de Fibonacci

Peut-on utiliser l'exponentiation rapide pour la suite de Fibonacci ? Oui si on met le problème sous la forme vectorielle $X_{n+1} = AX_n$

$$X_n = \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix}, \qquad A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \qquad X_n A^n X_0$$

Comparer la complexité

- de la méthode naïve,
- de la méthode vectorielle,
- de cette technique par exponentiation rapide