Plan du chapitre : introduction à la récursivité

1. Définition et exemple de base
   - Définition - Vocabulaire
   - Premiers exemples de base
   - Analyse d'un exemple : la suite de Fibonnaci
2. Intérêt et limitation de la récursivité
   - Consommation et performance
   - Facilité et lisibilité
3. Programmation récursive de la dichotomie
   - Programmation itérative
   - Réécriture de la fonction
   - Performances en temps de la fonction

Notion de récursivité

- Jusqu'à présent, lorsque nous avons écrit des fonctions elles étaient constitués d'une suite d'instructions qui devaient être exécutées à l'appel de la fonction avec, souvent, une itération d'un certain nombres d'entre elles à l'aide de boucles. Par exemple on peut ainsi programmer le calcul du factoriel d'un entier :

Notion de récursivité

- Jusqu'à présent, lorsque nous avons écrit des fonctions elles étaient constitués d'une suite d'instructions qui devaient être exécutées à l'appel de la fonction avec, souvent, une itération d'un certain nombres d'entre elles à l'aide de boucles. Par exemple on peut ainsi programmer le calcul du factoriel d'un entier :

```
def factoriel_iteratif(n : int) -> int :
    # n : type int
    # condition : n>0
    a = 1
    for k in range(n+1) :
        a = a*k
    return a
```

Notion de récursivité

- Jusqu'à présent, lorsque nous avons écrit des fonctions elles étaient constitués d'une suite d'instructions qui devaient être exécutées à l'appel de la fonction avec, souvent, une itération d'un certain nombres d'entre elles à l'aide de boucles. Par exemple on peut ainsi programmer le calcul du factoriel d'un entier :

```
def factoriel_iteratif(n : int) -> int :
    # n : type int
    # condition : n>0
    a = 1
    for k in range(n+1) :
        a = a*k
    return a
```

- Mais il serait possible aussi de penser au fait que : $(n+1)! = (n+1) \times n!$ et que donc, pour calculer le factoriel d'un entier la fonction va "s'appeler elle même" !
- En informatique, on dit qu'une fonction est récursive si l'une des instructions qu'elle contient fait appel à la fonction elle-même.

Premier exemple : I

- Reprenons l'exemple de la fonction factorielle qui est "naturellement" de ce type du fait que :

$$\underset{\text{fonction pour } n}{n!} = n \times \underset{\text{fonction pour } n-1}{(n-1)!}$$

- On donne ci-dessous l'exemple d'une fonction calculant $n!$ pour n entier naturel

```
def factoriel_recursif(n:int)->int :
    # cette condition assure que les
    # appels successifs vont s'arrêter.
    if n == 0 :
        return 1
    else :
        # la fonction va renvoyer le
        # résultat d'un appel à elle même
        return n*factoriel_recursif(n-1)
```

Premier exemple : II

- On notera la présence dans cette fonction de la condition de terminaison qui doit assurer que les appels à la fonction ne vont pas continuer indéfiniment.
- Si on notre condition de terminaison n'est pas correcte (ou absente) comme par exemple dans la fonction suivante :

```
def factoriel_recursif(n) :
    if n == 7 :
        return 1
    else :
        # la fonction va renvoyer le
        # résultat d'un appel à elle même
        return n*factoriel_recursif(n-1)
factoriel_recursif(5)
```

Python répond :

```
RecursionError: maximum recursion depth exceeded in comparis
```

A vous de joueur ...

Compléter la fonction suivante pour qu'elle calcule de manière récursive la somme des entiers naturels de 0 à n.

```
def somme_entier(n) :
    # condition de terminaison et retour
    if .... :
        return ...
    else :
        return ....
```

Suite de Fibonacci

On considère la suite dite de Finonaci définie par $u_0 = 0, u_1 = 1$ puis pour $n \in \mathbb{N}$ par :

$$u_{n+2} = u_{n+1} + u_n.$$

1. Écrire une fonction itérative `fibo_iter` prenant en paramètre un entier n et renvoyant la valeur du terme d'indice n de la suite de Fibonacci

Suite de Fibonacci

On considère la suite dite de Finonaci définie par $u_0 = 0, u_1 = 1$ puis pour $n \in \mathbb{N}$ par :

$$u_{n+2} = u_{n+1} + u_n.$$

1. Écrire une fonction itérative `fibo_iter` prenant en paramètre un entier n et renvoyant la valeur du terme d'indice n de la suite de Fibonacci

```
def fibo_iter(n : int)->int :
    u0,u1 = 0,1    #les deux derniers termes connus
    if n == 0 :
        return u0
    if n ==  1 :
        return u1
    for k in range(0,n-1) : # le terme d'indice n : n-1 opérat
        u1,u0 = u0+u1,u1
    return u1
```

Version Récursive

1. Programmer une version récursive de cette fonction de la fonction précédente : `fibo_recur`

Version Récursive

1. Programmer une version récursive de cette fonction de la fonction précédente : `fibo_recur`

```
def fibo_recur(n:int)->int :
    if n == 0 :
        return 0
    if n == 1 :
        return 1
    return fibo_recur(n-1)+fibo_recur(n-2)
```

Tests de temps

- Pour faire des tests de temps d'exécution d'une série de commandes, on peut employer le module `time` et sa fonction `time` ou sa fonction `perf_counter`

```
t1 = time.perf_counter()
commande1
commande2
...
commande_finale
temps_execution = time.perf_coutner()-t1
```

Tests de temps

- Pour faire des tests de temps d'exécution d'une série de commandes, on peut employer le module `time` et sa fonction `time` ou sa fonction `perf_counter`

```
t1 = time.perf_counter()
commande1
commande2
...
commande_finale
temps_execution = time.perf_coutner()-t1
```

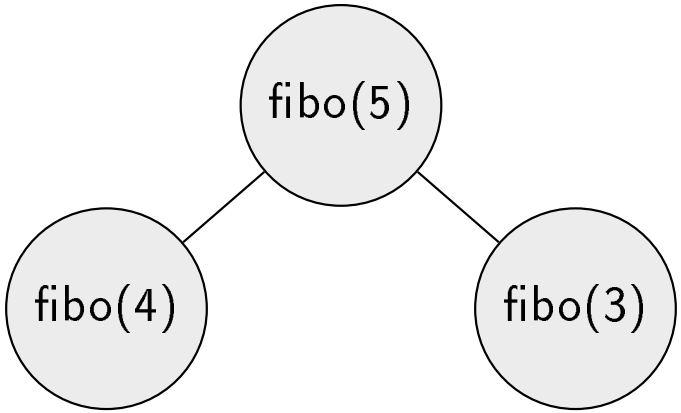
- On donne les résultats suivants pour les temps de calculs du terme d'indice 30 :
  - temps en itératif : 2.5686999833851587e-05
  - temps en récursif : 0.3736828729997796 5

Explication :

Dresser sur papier un arbre représentant les appels successifs à la fonction `fibo_recur(6)` lors de son exécution.

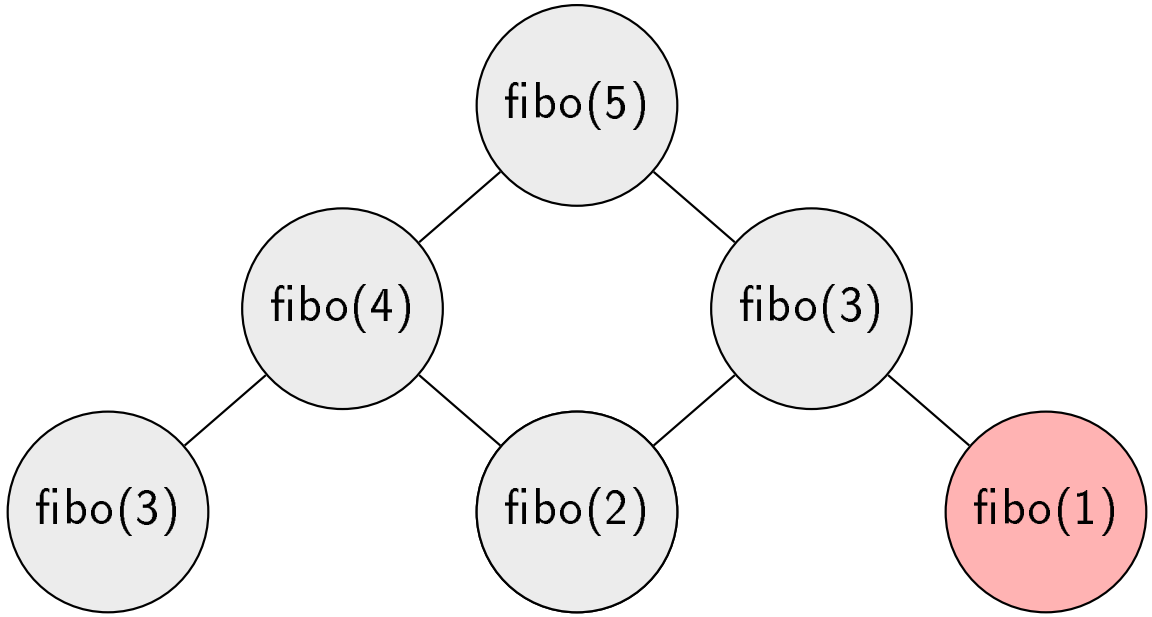
Explication :

Dresser sur papier un arbre représentant les appels successifs à la fonction `fibo_recur(6)` lors de son exécution.



Explication :

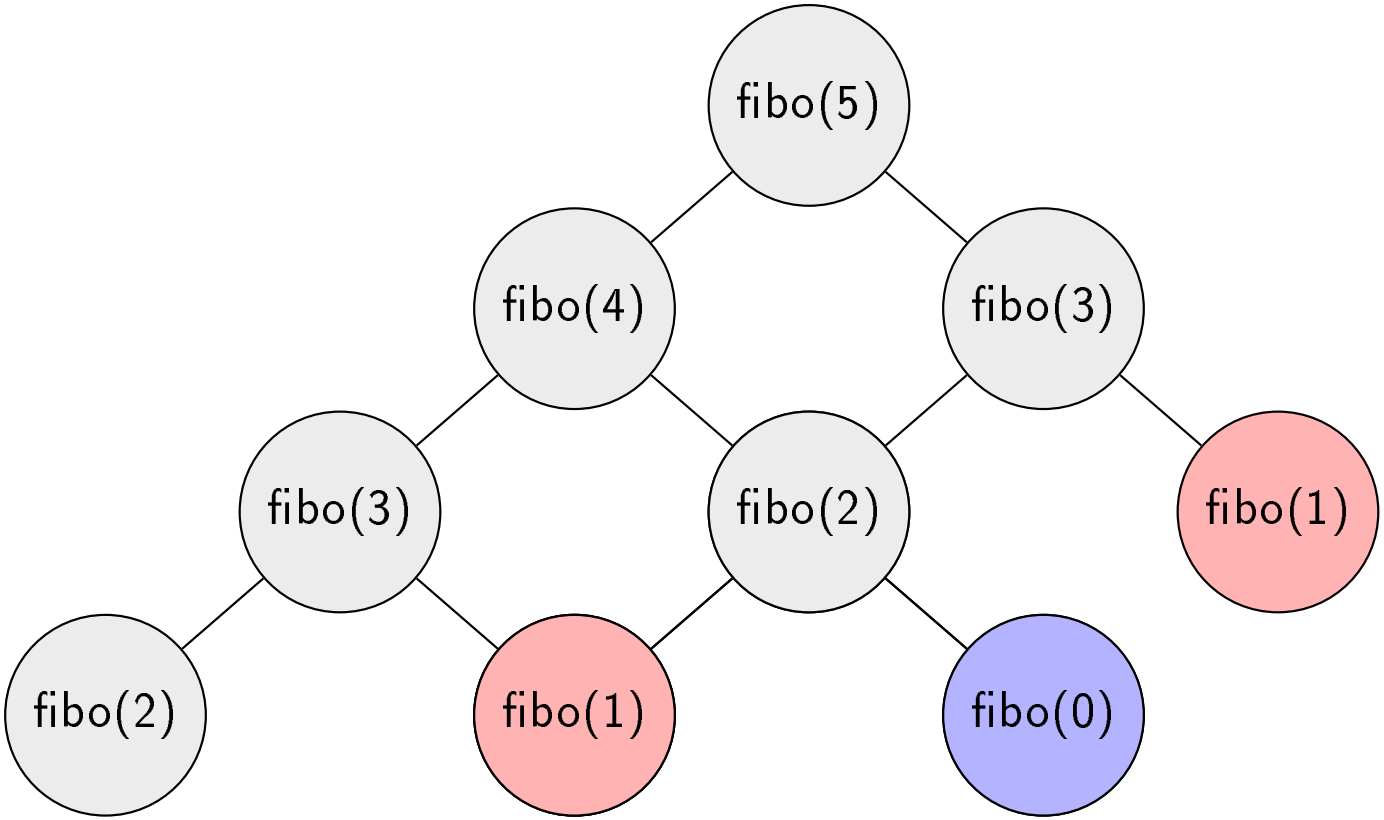
Dresser sur papier un arbre représentant les appels successifs à la fonction `fibo_recur(6)` lors de son exécution.



.... commentaires !

Explication :

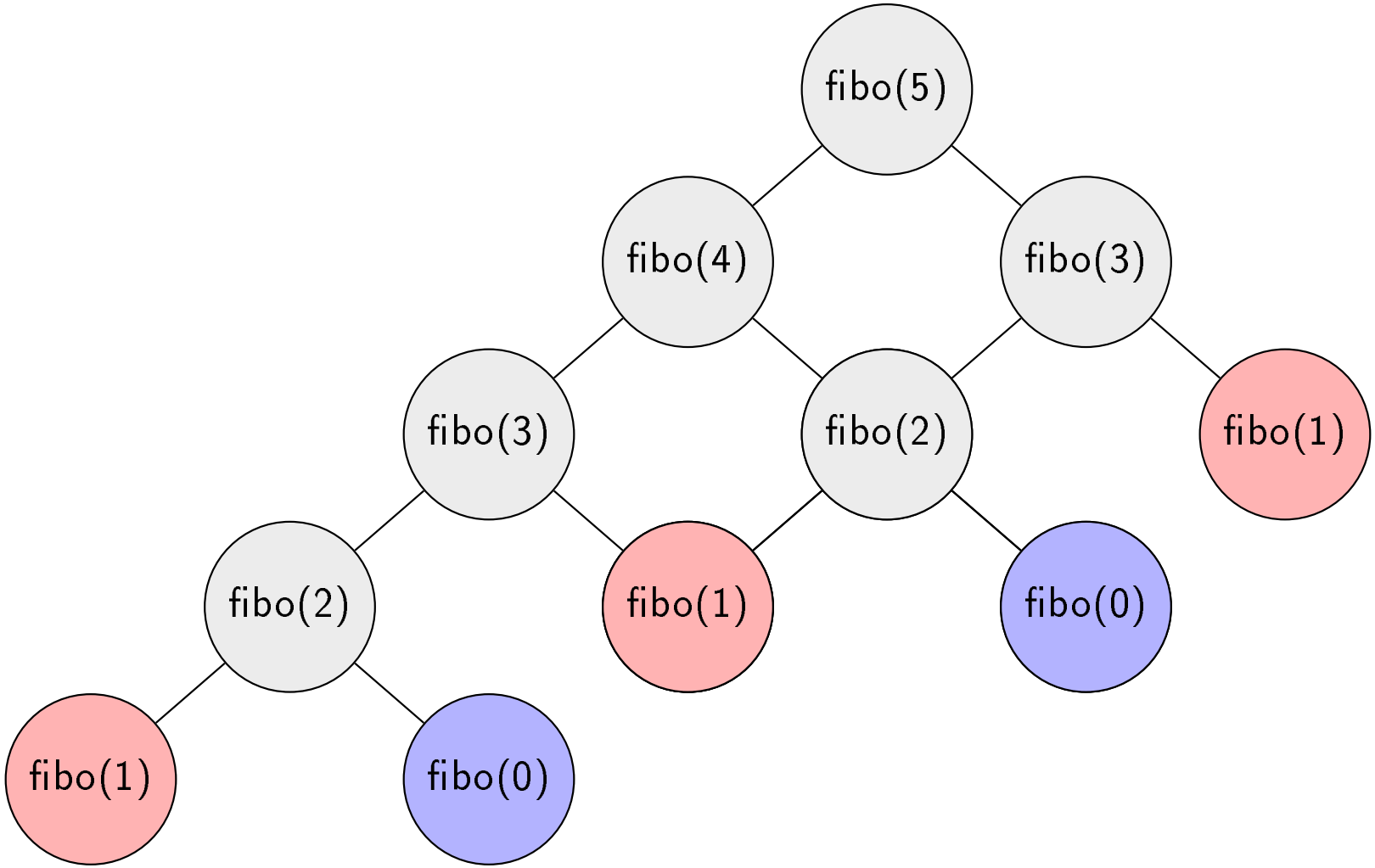
Dresser sur papier un arbre représentant les appels successifs à la fonction `fibo_recur(6)` lors de son exécution.



.... commentaires !

Explication :

Dresser sur papier un arbre représentant les appels successifs à la fonction `fibo_recur(6)` lors de son exécution.



.... commentaires !

Limitation de la pile d'appels

1. On a déjà vu que les performance en temps n'étaient pas forcément meilleures
2. L'autre problème est celui de la pile d'appel

Limitation de la pile d'appels

1. On a déjà vu que les performance en temps n'étaient pas forcément meilleures
2. L'autre problème est celui de la pile d'appel
3. Par exemple pour la fonction suivante qui calcule les termes d'une suite arithmético-géométrique :

```
def suite(a,b,u,n) :
    if n == 0 :
        return u
    else :
        # on relance la fonction a partir
        # de la valeur suivante de la suite
        # mais en calculant un terme de moins
        return suite(a,b,a*u+b,n-1)
```

4. L'appel suivant : `suite(-1,2,1,1000)` engendre l'erreur suivante :

```
RecursionError: maximum recursion depth exceeded in comparis
```

5. Qui signifie qu'on a dépassé le maximum d'appels successif de la fonction elle même : ce qu'on appelle la profondeur de la pile d'appels.
6. Alors que que la même fonction programmée de manière itérative

Intérêt

- On traitera en TP des exemples plus soutenus trop longs à traiter ici.
- Mais on peut souligner que dans certains cas, la récursivité donne une programmation beaucoup plus facile à écrire.
- Prenons par exemple le cas de la fonction de Ackerman qui est une fonction définie sur les couples d'entiers naturels (m, n) par :

$$A(m,n) = \begin{cases} n+1 & \text{si} \quad m = 0 \\ A(m-1,1) & \text{si} \quad m > 0 \quad \text{et} \quad n = 0 \\ A(m-1, A(m, n-1)) & \text{si} \quad m > 0 \quad \text{et} \quad n > 0 \end{cases}$$

- La programmation récursive est très naturelle :

Intérêt

- On traitera en TP des exemples plus soutenus trop longs à traiter ici.
- Mais on peut souligner que dans certains cas, la récursivité donne une programmation beaucoup plus facile à écrire.
- Prenons par exemple le cas de la fonction de Ackerman qui est une fonction définie sur les couples d'entiers naturels (m, n) par :

$$A(m,n) = \begin{cases} n+1 & \text{si} \quad m = 0 \\ A(m-1,1) & \text{si} \quad m > 0 \quad \text{et} \quad n = 0 \\ A(m-1, A(m, n-1)) & \text{si} \quad m > 0 \quad \text{et} \quad n > 0 \end{cases}$$

- La programmation récursive est très naturelle :

```
def A(m,n) :
    if m == 0  :
        return n + 1
    elif n == 0 :
        return A(m-1,1)
    else :
        return A(m-1,A(m,n-1))
```

Dichotomie en itératif

- Écrire une fonction itérative qui prend en argument les bornes `a`,`b` d'un intervalle une précision `p` et une fonction `f`, et qui renvoie une valeur approché à `p` près d'une solution de `f(x)=0` obtenue par dichotomie sur l'intervalle `[a,b]`.
- Syntaxe `dicho_iter(a,b,p,f)`

Dichotomie en itératif

- Écrire une fonction itérative qui prend en argument les bornes `a,b` d'un intervalle une précision `p` et une fonction `f`, et qui renvoie une valeur approché à `p` près d'une solution de `f(x)=0` obtenue par dichotomie sur l'intervalle `[a,b]`.
- Syntaxe `dicho_iter(a,b,p,f)`

```
def dicho_iter(a,b,p,f) :
```

Dichotomie en itératif

- Écrire une fonction itérative qui prend en argument les bornes `a,b` d'un intervalle une précision `p` et une fonction `f`, et qui renvoie une valeur approché à `p` près d'une solution de `f(x)=0` obtenue par dichotomie sur l'intervalle `[a,b]`.
- Syntaxe `dicho_iter(a,b,p,f)`

```
def dicho_iter(a,b,p,f) :
    while b-a> p :
```

Dichotomie en itératif

- Écrire une fonction itérative qui prend en argument les bornes `a,b` d'un intervalle une précision `p` et une fonction `f`, et qui renvoie une valeur approché à `p` près d'une solution de `f(x)=0` obtenue par dichotomie sur l'intervalle `[a,b]`.
- Syntaxe `dicho_iter(a,b,p,f)`

```
def dicho_iter(a,b,p,f) :
    while b-a> p :
        m = (a+b)/2
        val = f(m)*f(a)
```

Dichotomie en itératif

- Écrire une fonction itérative qui prend en argument les bornes `a,b` d'un intervalle une précision `p` et une fonction `f`, et qui renvoie une valeur approché à `p` près d'une solution de `f(x)=0` obtenue par dichotomie sur l'intervalle `[a,b]`.
- Syntaxe `dicho_iter(a,b,p,f)`

```
def dicho_iter(a,b,p,f) :
    while b-a> p :
        m = (a+b)/2
        val = f(m)*f(a)
        if val <= 0 : # f(m) et f(a) de signes différents
            b = m
```

Dichotomie en itératif

- Écrire une fonction itérative qui prend en argument les bornes `a,b` d'un intervalle une précision `p` et une fonction `f`, et qui renvoie une valeur approché à `p` près d'une solution de `f(x)=0` obtenue par dichotomie sur l'intervalle `[a,b]`.
- Syntaxe `dicho_iter(a,b,p,f)`

```
def dicho_iter(a,b,p,f) :
    while b-a> p :
        m = (a+b)/2
        val = f(m)*f(a)
        if val <= 0 : # f(m) et f(a) de signes différents
            b = m
        else : # f(m) et f(a) de même signe
        # donc f(m) de signe différent de f(b)
            a = m  # du coup on a encore f(a)*f(b)<= 0
```

Dichotomie en itératif

- Écrire une fonction itérative qui prend en argument les bornes a,b d'un intervalle une précision p et une fonction f, et qui renvoie une valeur approché à p près d'une solution de f(x)=0 obtenue par dichotomie sur l'intervalle [a,b].
- Syntaxe dicho_iter(a,b,p,f)

```
def dicho_iter(a,b,p,f) :
    while b-a> p :
        m = (a+b)/2
        val = f(m)*f(a)
        if val <= 0 : # f(m) et f(a) de signes différents
            b = m
        else : # f(m) et f(a) de même signe
        # donc f(m) de signe différent de f(b)
            a = m  # du coup on a encore f(a)*f(b)<= 0
    return xmin
```

Version récursive

- Mais en fait cette méthode de dichotomie est récursive dans le sens où on la recommence à chaque étape sur un nouvel intervalle plus petit ....
- Programmer une version récursive de la méthode de dichotomie.

Version récursive

- Mais en fait cette méthode de dichotomie est récursive dans le sens où on la recommence à chaque étape sur un nouvel intervalle plus petit ....
- Programmer une version récursive de la méthode de dichotomie.
- Cela donne :

```
def dichotomie(a,b,p,f) :
```

Version récursive

- Mais en fait cette méthode de dichotomie est récursive dans le sens où on la recommence à chaque étape sur un nouvel intervalle plus petit ....
- Programmer une version récursive de la méthode de dichotomie.
- Cela donne :

```
def dichotomie(a,b,p,f) :
    # condition de terminaison :
    # l'écart entre a et b est inférieur
    # à la précision p demandée.
```

Version récursive

- Mais en fait cette méthode de dichotomie est récursive dans le sens où on la recommence à chaque étape sur un nouvel intervalle plus petit ....
- Programmer une version récursive de la méthode de dichotomie.
- Cela donne :

```
def dichotomie(a,b,p,f) :
    # condition de terminaison :
    # l'écart entre a et b est inférieur
    # à la précision p demandée.
    if (b-a) <= p :
        return a,b
```

Version récursive

- Mais en fait cette méthode de dichotomie est récursive dans le sens où on la recommence à chaque étape sur un nouvel intervalle plus petit ....
- Programmer une version récursive de la méthode de dichotomie.
- Cela donne :

```
def dichotomie(a,b,p,f) :
    # condition de terminaison :
    # l'écart entre a et b est inférieur
    # à la précision p demandée.
    if (b-a) <= p :
        return a,b
    elif f(a)*f((a+b)/2)< 0 :
        return dichotomie(a,(a+b)/2,p,f)
```

Version récursive

- Mais en fait cette méthode de dichotomie est récursive dans le sens où on la recommence à chaque étape sur un nouvel intervalle plus petit ....
- Programmer une version récursive de la méthode de dichotomie.
- Cela donne :

```
def dichotomie(a,b,p,f) :
    # condition de terminaison :
    # l'écart entre a et b est inférieur
    # à la précision p demandée.
    if (b-a) <= p :
        return a,b
    elif f(a)*f((a+b)/2)< 0 :
        return dichotomie(a,(a+b)/2,p,f)
    else :
        return dichotomie((a+b)/2,b,p,f)
```

Mesures de temps d'exécutions

- Test fait sur ma machine pour 10000 répétition de la fonction précédente avec une précision demandée en 2^{-10} et avec la fonction :

$$x \mapsto x^2 - 2.$$

  - Temps avec la fonction écrite sous forme récursive : `0.07592725753784180`.
  - Temps avec la fonction écrite sous forme itérative : `0.05460977554321289`.
- Sur cet exemple il y aurait plutôt une perte de performance en temps.
- Mais ce n'est pas une question de nombre théorique d'opérations car il reste en : constante $\times k$ pour une précision de 2^{-k}. (on le démontrera précisément plus tard).

Retour sur le problème des calculs inutiles

- Écrire une version récursive du calcul des termes de la suite de Fibonacci mais cette fois en ajoutant dans votre fonction la mémorisation des éléments déjà calculés dans une liste ... et en faisant un retour directement si l'élément demandé est déjà dans la liste !

Retour sur le problème des calculs inutiles

- Écrire une version récursive du calcul des termes de la suite de Fibonacci mais cette fois en ajoutant dans votre fonction la mémorisation des éléments déjà calculés dans une liste ... et en faisant un retour directement si l'élément demandé est déjà dans la liste !
- La fonction aura la syntaxe : `fibo_recur(n,memo = None)` et elle renverra les résultats calculés dans la liste `memo`

```
def fibo(n, memo=None):
```

Retour sur le problème des calculs inutiles

- Écrire une version récursive du calcul des termes de la suite de Fibonacci mais cette fois en ajoutant dans votre fonction la mémorisation des éléments déjà calculés dans une liste ... et en faisant un retour directement si l'élément demandé est déjà dans la liste !
- La fonction aura la syntaxe : `fibo_recur(n,memo = None)` et elle renverra les résultats calculés dans la liste `memo`

```
def fibo(n, memo=None):
    # Création de la liste de mémorisation une seule fois
    if memo is None:
        memo = [0, 1] + [None] * (n - 1)
```

Retour sur le problème des calculs inutiles

- Écrire une version récursive du calcul des termes de la suite de Fibonacci mais cette fois en ajoutant dans votre fonction la mémorisation des éléments déjà calculés dans une liste ... et en faisant un retour directement si l'élément demandé est déjà dans la liste !
- La fonction aura la syntaxe : `fibo_recur(n,memo = None)` et elle renverra les résultats calculés dans la liste `memo`

```
def fibo(n, memo=None):
    # Création de la liste de mémorisation une seule fois
    if memo is None:
        memo = [0, 1] + [None] * (n - 1)
    # Si la valeur est déjà calculée, on la renvoie
    if memo[n] is not None:
        return memo[n]
```

Retour sur le problème des calculs inutiles

- Écrire une version récursive du calcul des termes de la suite de Fibonacci mais cette fois en ajoutant dans votre fonction la mémorisation des éléments déjà calculés dans une liste ... et en faisant un retour directement si l'élément demandé est déjà dans la liste !
- La fonction aura la syntaxe : `fibo_recur(n,memo = None)` et elle renverra les résultats calculés dans la liste `memo`

```
def fibo(n, memo=None):
    # Création de la liste de mémorisation une seule fois
    if memo is None:
        memo = [0, 1] + [None] * (n - 1)
    # Si la valeur est déjà calculée, on la renvoie
    if memo[n] is not None:
        return memo[n]
    # Sinon on calcule et on stocke dans memo
    memo[n] = fibo(n-2, memo) + fibo(n-1, memo)
    return memo[n]
```

Retour sur le problème des calculs inutiles

- Écrire une version récursive du calcul des termes de la suite de Fibonacci mais cette fois en ajoutant dans votre fonction la mémorisation des éléments déjà calculés dans une liste ... et en faisant un retour directement si l'élément demandé est déjà dans la liste !
- La fonction aura la syntaxe : `fibo_recur(n,memo = None)` et elle renverra les résultats calculés dans la liste `memo`

```
def fibo(n, memo=None):
    # Création de la liste de mémorisation une seule fois
    if memo is None:
        memo = [0, 1] + [None] * (n - 1)
    # Si la valeur est déjà calculée, on la renvoie
    if memo[n] is not None:
        return memo[n]
    # Sinon on calcule et on stocke dans memo
    memo[n] = fibo(n-2, memo) + fibo(n-1, memo)
    return memo[n]
# Exemple d'utilisation
print(fibo(35))
```