Dans ce cours nous allons reprendre la récursivité simple pour l'améliorer en programmation dynamique lorsque cela sera utile.

1 Récursivité simple :

1 . Définitions :

Une fonction récursive est une fonction dont la définition contient un (ou plusieurs) appel à elle-même.

Point terminal : comme dans le cas d'une boucle, il faut un cas d'arrêt où l'on ne fait pas d'appel récursif. C'est le point terminal.

Algorithme 1 : Procédure récursive (paramètres)

```
si TEST D'ARRET alors
    instructions du point d'arrêt
sinon
    instructions
    récursive(nouveaux paramètres) ; // appel récursif
    instructions
```

Notion d'ordre bien fondé De manière générale : Pour prouver qu'une fonction récursive f termine sur son domaine de définition - $\forall$ x $\in$ Df , $\exists y$: f (x) = y

- Déterminer son domaine de définition
- Exhiber un ordre (strict) bien fondé sur le domaine de définition
- Montrer que les arguments des appels récursifs restent dans le domaine de définition de la fonction et sont plus petits (au sens de l'ordre bien fondé) que l'argument de l'appel initial.

Exemple de la fonction factorielle : $n! = n \times (n-1) \times ..... \times 2 \times 1$ où $n \in \mathbb{N}$

Elle est définie par $0! = 1$ et pour tout $n > 0 \rightarrow n! = n \times (n-1)!$

Plusieurs types d'algorithmes permettent de calculer cette fonction :

Algorithme 2 : version itérative - fact_iter (n)

```
entier i, resultat ;
resultat ←1 ;
si n > 0 alors
    pour i allant de 1 à n par pas de 1 faire
        resultat ← resultat× i ;
retourne resultat ;
```

Algorithme 3 : version récursive - fact_rec (n)

```
si n = 0 alors
    retourne 1 ;
sinon
    retourne n × fact_rec (n-1) ;
```

NB1 : la fonction mathématique factorielle n'est ni récursive, ni itérative, c'est l'algorithme utilisée pour la calculer qui est l'un ou l'autre.

NB2 : le site http ://pythontutor.com/ permet de visualiser la trâce d'exécution de vos programmes. N'hésitez pas à voir ce qui se passe à chaque itération.

2 . Principe et dangers de la récursivité

Principe et intérêt :

Ce sont les mêmes que ceux de la démonstration par récurrence en mathématiques. On doit avoir :

— un certain nombre de cas dont la résolution est connue, ces « cas simples » formeront les cas d'arrêt de la récursion ;

— un moyen de se ramener d'un cas « compliqué » à un cas « plus simple ».

La récursivité permet d'écrire des algorithmes concis et élégants. Elle permet de s'adapter facilement à certaines structures de données comme les arbres par exemple ou de traiter aisément des problèmes de suites réccurentes.

Difficultés : Il faut être sûr que l'on retombe toujours sur un cas connu, c'est-à-dire sur une condition d'arrêt, il faut nous assurer que la fonction est complètement définie, c'est-à-dire, qu'elle est définie sur tout son domaine d'application.

L'écriture à l'aide d'une fonction récursive donne souvent un code plus lisible et plus susceptible d'être correct (car d'invariant plus simple) que son équivalent itératif utilisant une(des) boucle(s).

3 . Toute boucle for peut se transformer en une fonction récursive

Principe de passage d'itératif au récursif :

— Pour faire des choses pour un indice allant de 1 à n

— On les fait de 1 à n-1 (même traitement avec une donnée différente)

— Puis on les fait pour l'indice n (cas particulier)

Algorithme 4 : version itérative :

```
1 entier n ;
2 pour i allant de 1 à n par pas de 1 faire
3 |  traiter(i) ;
```

Algorithme 5 : Version récursive - entier n en entrée :

```
1 si n=0 alors
2 |  traiter(0)
3 sinon
4 |  recursive (n-1)
5 |  traiter(n)
```

4 . Pile d'exécution

Définition :

La Pile d'exécution du programme en cours est un emplacement mémoire destiné à mémoriser les paramètres, les variables locales ainsi que les adresses de retour des fonctions en cours d'exécution. Elle fonctionne selon le principe LIFO (Last-In-First-Out) : dernier entré premier sorti.

La version récursive utilise donc une pile LIFO. Attention ! La pile à une taille fixée, une mauvaise utilisation de la récursivité peut entraîner un débordement de pile (déjà vu en python !)

Voici par exemple ce qui se passe lors du calcul de 4 ! :

Phase descente	**Phase remontée**				
Fact_rec	n	n-1	Critère d'arrêt : n=0	n Fact_rec (n-1)	Fact_rec(n)
					24
Appel 1	4	3	Faux	24	6
Appel 2	3	2	Faux	6	2
Appel 3	2	1	Faux	2	1
Appel 4	1	0	Faux	1	1
Appel 5	0		Vrai		

2 La récursivité terminale

1 . Définitions : Dans la version dite récursive enveloppée du calcul de factorielle, pour calculer le produit $n \times fac(n-1)$, il faut préalablement obtenir la valeur de $fac(n-1)$ et il faut donc mettre cette multiplication en suspens pendant le calcul de $fac(n-1)$. Les appels sont ainsi emboîtés et la mémoire nécessaire croît à chaque appel. Ainsi, l'exécution utilise un espace linéaire en n alors que l'algorithme itératif utilise un espace constant.

Il existe un type d'appel récursif qui n'est pas enveloppé. Grâce à un accumulateur on peut sortir l'appel récursif du calcul qui l'enveloppe. L'accumulateur sert à mémoriser le résultat des calculs intermédiaires au cours des différents appels. Dans ce cas aussi, l'algorithme utilise un espace mémoire constant.

La fonction factorielle peut être redéfinie à l'aide d'un accumulateur de cette manière :

Algorithme 6 : Fact-term (n, acc)

```
1 entier n ;
2 si n=0 alors
3 |  retourner (acc) // au lancement acc vaut 1 c'est à dire 0 !
4 sinon
5 |  retourner (Fact-term (n-1, n× acc))
```

Ce type d'appel récursif est appelé "terminal", car l'appel récursif n'est plus imbriqué dans aucun calcul.

2 . Définition : Une fonction récursive f est dite récursive terminale (**tail recursive**) si l'appel récursif est la dernière opération à effectuer lors de l'exécution du corps de la fonction. Tout appel récursif est de la forme retourner fonction récursive(...) , autrement dit, la valeur retournée est directement la valeur obtenue par un appel récursif, sans qu'il n'y ait aucune opération sur cette valeur.

C'est le cas de la fonction Fact-term. Ainsi, dans le cas d'une fonction récursive terminale, le dépilement des valeurs de retour est direct, il n'y a rien à retenir sur la pile. Ceci aboutit à une version plus optimisée de la fonction.

Dans une fonction récursive terminale les calculs de résultats intermédiaires sont réalisés avant l'appel récursif de la fonction. Au premier appel de la fonction on fournit la valeur initiale de l'accumulateur.

Pour le calcul de 4! avec la fonction récursive terminale ci-dessus : Fact-term(4,1)

Phase descente				
Fact-term	n	acc	n-1	n×acc
Appel 1	4	1	3	4
Appel 2	3	4	2	12
Appel 3	2	12	1	24
Appel 4	1	24	0	24
Appel 5	0	24		

On constate bien ici qu'il n'y a pas besoin de phase de remontée.

Initialisation : Dans le cas d'une fonction récursive terminale, il y a donc besoin d'initialiser l'accumulateur lors du lancement de la fonction. Il sera donc préférable afin de rendre cela transparent pour l'utilisateur, de créer une fonction principale qui appelle la fonction récursive terminale en en fixant les paramètres d'accumulation initiaux.

Par exemple pour la fonction Fact-term :

Fact-termL (n) Retourner Fact-term (n, 1)

Il est également possible afin de « masquer » la fonction récursive terminale de l'inclure dans une fonction principale.

Par exemple dans le cas de la fonction factorielle :

```
1 # Fact-termL ( n)
2 def Fact-term ( n, acc):
3     if (n=0) :
4         retourner (acc) \# au lancement acc vaut 1  c est     dire 0!
5     elif :
6         return Fact-term (n-1, n acc)
7     else :
8         return Fact-term ( n, 1)
```

L'inclusion de la fonction récursive terminale dans la fonction principale permet dans certains cas d'éliminer dans la fonction terminale les paramètres qui restent inchangés lors des appels successifs. Voir par exemple dans le III le a de la fonction puissance ou le x de la fonction racine.

3 . Toute fonction récursive peut être transformée en fonction récursive terminale :

Il suffit de stocker tous les paramètres intermédiaires lors de la descente pour éviter ainsi la remontée.

- La fonction doit avoir de bonnes propriétés (associativité, commutativité, ...)
- méthode : n opérations lors de la remontée ⇒ n paramètres supplémentaires
- il faut vérifier : que l'on peut trouver le résultat de cette manière que l'algorithme obtenu est récursif terminal

4 . Toute fonction récursive terminale peut être transformée en fonction itérative :

Il est effectivement facile de rendre une fonction récursive terminale en une fonction non récursive. Il suffit de simuler la récursivité par une boucle tant que. Cela n'utilise pas de mémoire supplémentaire. La négation du test d'arrêt de la récursivité est utilisée pour le test de la boucle tant que.

Algorithme 7 : factorielle(n) itératif

```
entier n ;
acc = 1 ;
tant que n ≠ 0 faire
    acc=acc×n;
    n ← n-1
; retourner acc ;
```

Voici par exemple, la fonction itérative fact :

D'une manière générale, pour rendre itérative une fonction récursive terminale :

Algorithme 8 : Algorithme récursif terminal

```
f(x, acc) :
si condition du point d'arrêt (x) alors
    instruction du point d'arrêt (x)
sinon
    T(x)
    r ← f(iteration suivante de x, g(x, acc))
```

Algorithme 9 : L'algorithme itératif suivant est équivalent :

```
f(x) :
u ← x tant que la condition du point d'arrêt ( u) est fausse faire
    T(u)
    u ← iteration suivante de u = h(u)
instruction du point d'arrêt (u)
```

5 . Exemples en Python :

– a – Somme des n premiers entiers :

```
# version recursive :
"""sigma:int->int
renvoie la somme des n premiers entiers naturels"""
def sigma(n):
    if n==0:
        return 0
    else :
        return n+sigma(n-1)
```

```
# version recursive terminale :
"""sigma_term:int->int
renvoie la somme des n premiers entiers naturels accumulateur a initialise a 0"""
def sigma_term(n,a):
    if n==0:return a
    else:return sigma_term(n-1,n+a)
```

– b – Calcul du PGCD :

Cette fonction est par construction récursive terminale :

```
# version recursive terminale :
"""pgcd(a,b):int,int->int
renvoie le pgcd de a et b"""
def pgcd(a,b):
    if b==0:    return a
    elif a==0:  return b
    elif a<b:   return pgcd(a,b-a)
    else:       return pgcd(a-b,b)
```

– c – Puissance naïve :

```
# version recursive
"""puisn:int,int->int
cette fonction calcule a puissance b en recursif naif"""
def puisn(a,b):
    if b==0:    return 1
    else:       return a*puisn(a,b-1)
```

```
# version recursive terminale
"""puisn_term :int,int,int->int
cette fonction calcule a puissance b en recursif terminal naif, acc initialise a 1"""
def puisn_term(a,b,acc):
    if b==0:    return acc
    else:       return puisn_term(a,b-1,a*acc)
```

– d – Puissance efficace :

```
# version recursive
"""puisrr:int->int
cette fonction calcule a puissance b en recursif algo rapide"""
def puisrr(a,b):
    if (b>1):
        if (b%2==0):
            return((puisrr(a,b/2))**2)
        else :
            return(a*(puisrr(a,(b-1)/2))**2)
    else:
       if (b==0):return(1)
       else:return(a)
```

```
# version recursive terminale
"""puisrr_term:int,int,int->int
cette fonction calcule a puissance b en recursif terminal algo rapide, accumulateur a
def puisrr_term(a,b,acc):
    if (b>1):
        if (b%2==0):
            return((puisrr_term(a,b/2,acc**2)))
        else :
            return((puisrr_term(a,(b-1)/2,a*acc**2)))
    else:
```

```
11          if (b==0):return 1
12          else:return(acc)
```

– e – Racine carrée

```
1 # version recursive
2 """racine:float->float
3 calcule une valeur approchee de la racine carree d'un nombre positif x au rang n"""
4 def racine(x,n):
5     if x<0:print("%f n'a pas de racine carree")%x
6     elif n==0:return 1
7     else:return (racine(x,n-1)+x/racine(x,n-1))/2.0
```

```
1 # version recursive terminale :
2 """racine:float,int,int->float
3 calcule une valeur approchee de la racine carree d'un nombre positif x au rang n
4 recursif terminal, a initialise a 1"""
5 def racine_term(x,n,a):
6     if x<0:print("%f n'a pas de racine carree") %x
7     elif n==0:return a
8     else:return racine_term(x,n-1,(a+x/a)/2.0)
```

– f – Fibonacci

```
1 # version recursive :
2 """fibo:int->int
3 calcul du nieme terme de la suite de Fibonacci"""
4 def fibo(n): #fibonacci en recursif
5     if n<=1: return n
6     else:return fibo(n-1)+fibo(n-2)
```

```
1 # version recursive terminale :
2 """fibo:int,int,int->int
3 calcul du nieme terme de la suite de Fibonacci, a initialise a 0 et b a 1"""
4 def fibo_term(n,a,b): #fibonacci en recursif terminal
5     if n==0: return a
6     elif n==1: return b
7     else:return fibo_term(n-1,b,a+b)
```

3 La programmation dynamique :

Visionner la vidéo de Cédric Gerland qui explique cet algorithme glouton en programmation dynamique :
"Programmation Dynamique avec Python Épisode 2 - Le rendu de monnaie"

```
def rendu_monnaie_rec(pieces:list[int],a_rendre:int)->int:
    """fonction de calcul de pieces a rendre par recursivite force brute"""
    if a_rendre == 0 :
        #Cas de base : aucune piece
        return 0
    else:
        #initialisation avec une valeur mini inatteignable
        mini = 9999
        for i in range(len(pieces)):
            #On regarde pour chaque piece choisie le nb mini possible
            if pieces[i]<= a_rendre:
                nb = 1 + rendu_monnaie_rec(pieces, a_rendre[i])
                #Nb pieces = LA piece+le nb de pieces restantes
                if nb < mini:
                    #Si un nouveau mini alors on memorise dans mini
                    mini = nb
    return mini

def rendu_monnaie_dynamique(pieces:list[int],a_rendre:int,rendu_memo:list[int])->int:
    """programmation dynamique du rendu de monnaie
    avec memorisation des possibilites deja traitees"""
    if a_rendre == 0 :
        #Cas de base : aucune piece
        return 0

    elif rendu_memo[a_rendre] > 0:
        #cas du nb de pieces deja calcule, on renvoie ce nb
        return rendu_memo[a_rendre]

    else:
        #initialisation avec une valeur mini inatteignable
        mini = 9999
        for i in range(len(pieces)):
            #
            if pieces[i] <= a_rendre:
                nb = 1+rendu_monnaie_dynamique(pieces,a_rendre[i],rendu_memo)
                if nb < mini:
                    #Si un nouveau mini alors on memorise dans mini
                    #Et, on stocke ce rendu pour une nouvelle utilisation
                    mini = nb
                    rendu_memo[a_rendre] = mini
    return mini
if __name__=='__main__':
    a_rendre=45
    rendu_memo = [0] * (a_rendre + 1)
    pieces=[1,2,5,10,20,100,200]

    nb_pieces = rendu_monnaie_dynamique(pieces, a_rendre, rendu_memo)
    print(nb_pieces)
```

Source Wikipédia : En informatique, on parle de programmation dynamique par mémoïsation : c'est la mise en cache des valeurs de retour d'une fonction selon ses valeurs d'entrée. Le but de cette technique d'optimisation de code est de diminuer le temps d'exécution d'un programme informatique en mémorisant les valeurs retournées par une fonction. Bien que liée à la notion de cache, la mémoïsation désigne une technique bien distincte de celles mises en œuvre dans les algorithmes de gestion de la mémoire cache.

La programmation dynamique est une technique algorithmique utilisée pour résoudre des problèmes en les décomposant en sous-problèmes plus simples, en calculant et en stockant les résultats de ces sous-problèmes, puis en les réutilisant au besoin pour éviter de recalculer plusieurs fois les mêmes valeurs. Cette technique est souvent utilisée pour optimiser les performances des algorithmes en évitant la répétition de calculs inutiles.

La programmation dynamique et la programmation récursive terminale sont deux concepts différents en informatique, bien qu'ils puissent parfois être utilisés ensemble pour résoudre certains problèmes.

D'autre part, la programmation récursive terminale est une forme spécifique de récursivité où les appels récursifs sont effectués à la fin de la fonction et où aucune opération n'est effectuée après l'appel récursif. Dans ce cas, les appels récursifs peuvent être remplacés par une boucle itérative, ce qui permet d'éviter la consommation excessive de la pile d'appels récursifs.

Voici un exemple classique de programmation dynamique : le calcul du n-ième terme de la suite de Fibonacci. La suite de Fibonacci est définie de la manière suivante :

Algorithme 10 : Suite de Fibonacci entier n en entrée

```
1 F(0)=0
2 F(1)=1
3 F(n)=F(n-1)+F(n-2) pour n>=2
```

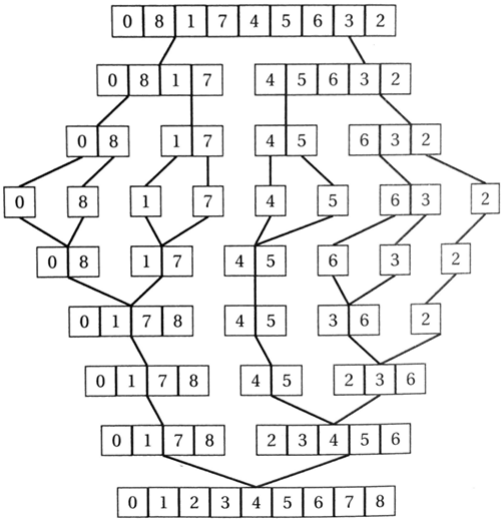
Voici un exemple de code en Python utilisant la programmation dynamique pour calculer le n-ième terme de la suite de Fibonacci :

```
def fibonacci(n):
    # Initialiser un tableau pour stocker les valeurs de Fibonacci
    fib = [0] * (n + 1)

    # Definir les cas de base
    fib[0] = 0
    fib[1] = 1

    # Calculer les termes de la suite de Fibonacci
    for i in range(2, n + 1):
        fib[i] = fib[i - 1] + fib[i - 2]

    # Retourner le terme recherche
    return fib[n]

# Exemple d'utilisation
n = 10
print(f"Le {n}-ieme terme de la suite de Fibonacci est : {fibonacci(n)}")
```

Dans ce code, nous utilisons un tableau fib pour stocker les valeurs de Fibonacci déjà calculées. Nous initialisons ce tableau avec les valeurs de base de la suite (0 et 1), puis nous parcourons le reste des valeurs jusqu'à atteindre le n-ième terme. En utilisant la programmation dynamique de cette manière, nous évitons de recalculer les valeurs déjà connues et réduisons ainsi la complexité temporelle de l'algorithme.

4 Le tri fusion :

La méthode est la suivante :



Je vous propose ce programme en python :

```
from random import randint
l=[]
for i in range (11):
    l.append(randint(-8,10))
print (l)

def triFusion (l):
    n=len(l)
    if n==0 or n==1:
        return l
    else:
        return fusion(triFusion(l[0:n//2]),triFusion(l[n//2:n]))

def fusion (l1,l2):
    '''␣entr e␣couple␣de␣listes␣tri es
␣␣␣␣␣␣␣␣sortie␣une␣liste␣tri e␣par
␣␣␣␣␣␣␣␣ordre␣croissant'''
    if l1==[]:
        return l2
    elif l2==[]:
        return l1
    else :
        i=0
        j=0
        res=[]
        while ((i<len(l1) )and (j<len(l2))):
            if (l1[i] < l2[j]):
                res.append (l1[i])
                i += 1

            else :
                res.append (l2[j])
```

```
            j += 1
        if (i>=len (l1)):
            res=res+(l2[j:len(l2)])
        else :
            res=res+(l1[i:len(l1)])
        return res
print ( triFusion(l))
```

Le tri fusion est un tri optimal sur les listes, de complexité O(n·ln(n)). Il s'agit de décomposer une liste en deux sous-listes chacune deux fois plus petites, de les trier séparément, puis de fusionner les résultats en une liste triée.