Thème 1 : la récursivité

« To understand recursion, you must understand recursion. »
Inconnu

1 Rappels sur les fonctions

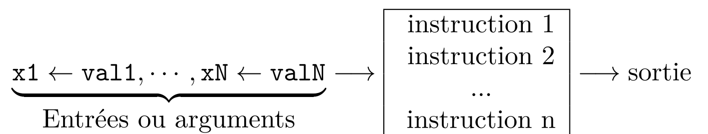
1.1 Qu'est-ce qu'une fonction ?

Définition. Une fonction est une séquence d'instructions que l'on regroupe sous un même nom ; en général une fonction comporte des variables ou *paramètres d'entrée*.
L'appel de la fonction consiste en l'exécution de la séquence d'instruction pour des valeurs des paramètres d'entrée (arguments) ; cet appel retourne une valeur de sortie.

Pour définir nos fonctions nous utiliserons du pseudo-code ou la syntaxe du langage Python rappelée ci-dessous.

Définition de la fonction :

```
def maFonction(x1,...,xN) :
    """commentaires"""
    instruction 1
    ...
    instruction n
    return sortie
```



Appel de la fonction :

```
maFonction(val1,...,valN)
```

Exercice 1. On définit une fonction `puissance2(n)` dont le paramètre est un entier n et qui retourne 2^n.

Pseudo-code	Python

Fonction : puissance2(n).
Entrée : un entier n
Sortie : l'entier 2^n
$k = 0$
$p = 1$
Tant que $k < n$:
    $p = 2p$
    $k = k + 1$
Retourner p	

Justifier la terminaison de la fonction ci-dessus. Justifier qu'elle est correcte.

1.2 Variables locales et variables globales

Définition. Les variables utilisées dans le corps d'une fonction sont des **variables locales**. Elles sont affectées au moment de l'appel de la fonction et sont détruites à la sortie. Les paramètres d'entrés subissent le même traitement.

On peut définir des fonctions et utiliser des variables locales dans ces fonctions sans se préoccuper de savoir si elles existent ailleurs dans le script. Les affectations temporaires à l'appel de la fonction ne risquent pas d'interférer avec les autres variables.

Les variables utilisées à l'extérieur d'une fonction sont des **variables globales**. Une fonction peut modifier une variable globale à condition de la déclarer comme telle dans le corps de la fonction : on utilise l'instrution `global`.

1.3 Fonctions et procédures en Python

Fonctions et procédures ont la même syntaxe ; on les distinguent de la manière suivante :

- Une fonction prend des arguments et retourne une valeur ;
- Une procédure réalise une action : affichage, écriture d'un fichier, ouverture d'une fenêtre graphique, modification d'un objet... Ces changements perdurent après l'appel de la procédure. Elle retourne la valeur `None`.

Exercice 2.
Écrire une procédure Python `echange(L,i,j)` dont les paramètres sont : `L` de type list, `i` et `j` de type `int`. La procédure échange `L[i]` et `L[j]`.

On peut naturellement envisager des objets de type mixte. D'un point de vue théorique nous travaillerons essentiellement avec des *fonctions pures.*

1.4 Exercice : la factorielle itérative

1. Écrire une fonction `fact(n)` dont l'argument d'entrée est un entier n. La fonction retourne la valeur $n!$ calculée à l'aide d'une boucle `for`.
2. Étudier la terminaison et la correction de votre fonction.

2 Fonctions récursives

2.1 Généralités

Définition. Une *fonction récursive* est une fonction qui contient au moins un appel à elle-même.

Un langage récursif est un langage dans lequel on peut programmer des fonctions récursives. Python est un langage récursif.

Exemple. Voici une définition récursive pour fonction `puissance2(n)` dont le paramètre est un entier n et qui retourne 2^n. On part du constat que :

$$\text{puissance2}(n) = 2^n = 2 \times 2^{n-1} = 2 \times \text{puissance2}(n-1)$$

Pseudo-code

Fonction : puissance2(n).

Entrée : un entier n

Sortie : l'entier 2^n

Si $n == 0$:

  Retourner 1

Sinon :

  Retourner $2\times$puissance2($n-1$)

Python

Que se passe-t'il lors de l'appel de `puissance2(n)` ?

Pour s'assurer de la terminaison d'une fonction récursive, on doit respecter les règles suivantes :

Règles.

- La fonction doit contenir un ou plusieurs *cas de base* ne comportant pas d'appel récursif.
  Dans la fonction `puissance2` : c'est le cas $n = 0$.
- Les appels de la fonction se font des arguments plus simple pour conduire aux cas de base.
  Dans l'appel de `puissance2(n)` : les arguments successifs sont $n-1 > n-2 > \cdots > 0$.

Voici deux mauvaises versions de la fonction `puissance2(n)`. **Pourquoi ?**

Mauvaise version 1.

```
def puissance2(n) :
    return 2*puissance2(n-1)
```

Mauvaise version 2.

```
def puissance2(n) :
    if n==0 :
        return 1
    else :
        return 4*puissance2(n-2)
```

2.2 Exemples simples de fonctions récursives

2.2.1 La factorielle récursive

Exercice 3. Écrire une fonction récursive `factRec(n)` pour le calcul de $n!$. Donner le nombre d'appel et le nombre de multiplication pour le calcul de $n!$.

2.2.2 L'exponentiation rapide

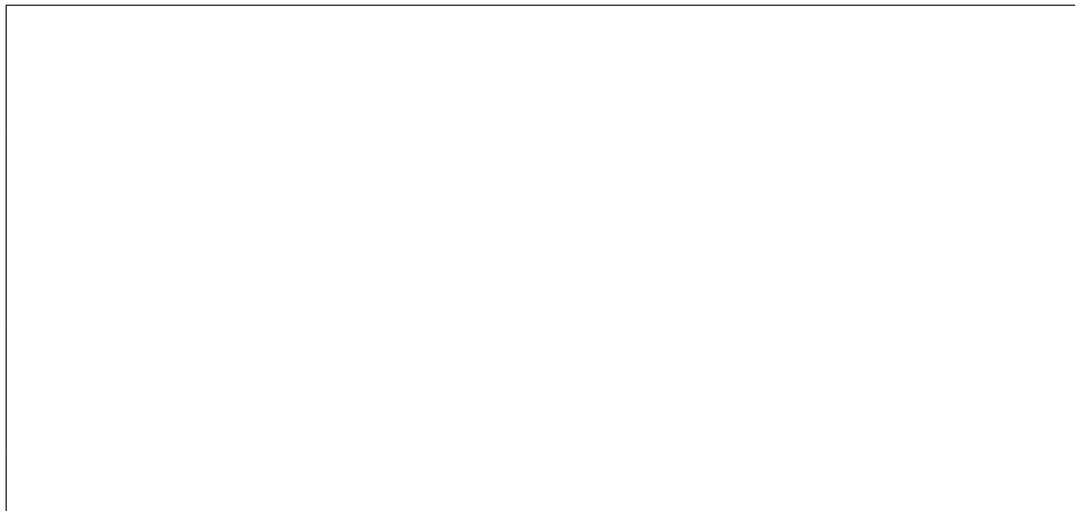
L'exponentiation rapide s'appuie sur un algorithme que nous présentons dans le cas particulier du calcul de q^{25}.

- Pour calculer q^{25}, on calcule $q^{12} \times q^{12} \times q$.
- Pour calculer q^{12}, on calcule $q^6 \times q^6$.
- Pour calculer q^6, on calcule $q^3 \times q^3$.
- Pour calculer q^3, on calcule $q^2 \times q$
- Pour calculer q^2, on calcule $q \times q$.
- Pour calculer q, on calcule $q \times 1$.

Nombre de multiplication nécessaires :

Écrire une fonction Python récursive `expo(q,n)` qui calcule q^n en s'appuyant sur le principe :

$$q^n = \begin{cases} \left(q^{\frac{n}{2}}\right)^2 & \text{si n est pair} \\ q.\left(q^{\frac{n-1}{2}}\right)^2 & \text{sinon.} \end{cases}$$



Exercice 4. Analyse de l'algorithme.

1. Justifier qu'un appel de `expo(q,n)` se termine.
2. Justifier qu'un appel de `expo(q,n)` retourne toujours q^n.
3. **Complexité de l'algorithme.** Notons $C(n)$ le nombre de multiplications lors d'un appel de `expo(q,n)` pour $n \geq 1$.
   (a) Justifier que $C(n) = C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + r_n$ avec r_n à expliciter en fonction de n.
   (b) Justifier que $\log_2(n) \leq C(n) \leq 2\log_2(n) + 1$. Comment qualifier la complexité de la fonction ?
   (c) Comparer sous Python avec la fonction `time` le temps d'exécution de `puissance(2,n)` et `expo(2,n)` pour de grandes valeurs de n.

2.3 La suite de Fibonacci : limite de la récursivité

La multiplication des lapins. Vous allez faire l'acquisition d'un couple de bébés lapins. Au bout d'un mois ce couple est adulte. Le mois suivant il donne naissance à un couple de bébés lapins : vous avez maintenant 4 lapins. Puis chaque couple engendre tous les mois un nouveau couple deux mois après sa naissance.

Nous avons le schéma ci-contre :

Légende : m : *bébé lapin* ; M : *lapin adulte*.

- **Mois 0.** m m
- **Mois 1.** M M
- **Mois 2.** M M m m
- **Mois 3.** M M M M m m
- **Mois 4.** M M M M M M m m m m

Notons F_N le nombre de lapins que l'on a au bout du N-ième mois. On convient que : $F_0 = 2$. Nous avons donc $F_1 = 2$ puis $F_2 = 4$ et $F_3 = 6$. Plaçons nous au mois $N + 2$, nous aurons tous les couples de lapins du mois précédent (le mois $N + 1$) et toutes les progénitures des couples de lapins du mois N. Nous avons donc la relation :

$$F_{N+2} = F_{N+1} + F_N$$

Écrire deux fonctions Python `FiboIt(N)` et `FiboRec(N)` pour le calcul de F_N. La première est rédigée de manière itérative, la seconce récursive.

`FiboIt(N)`	`FiboRec(N)`

__Exercice__ **5.**

1. **Analyse de l'algorithme itératif.** Estimer la complexité de l'algorithme itératif.
2. **Analyse de l'algorithme récursif.**
   (a) Dénombrer les appels de la fonction `FiboRec(N)` pour l'argument $N = 5$.
   (b) On note $A(N)$ le nombre d'appels récursifs de `FiboRec(N)`. Justifier que pour $N \geq 3$, $A(N) \geq \left(\frac{3}{2}\right)^N$.
   Comment qualifier cette complexité ?

3 Récursivité et tableaux : une première approche

« divide ut regnes. »
Jules César, Napoléon...

Dans cette partie on reprend quelques éléments d'algorithmique sur les listes (ou tableaux). On met en évidence la stratégie du *diviser pour régner* qui cherche à résoudre un problème en appliquant les trois principes suivants :

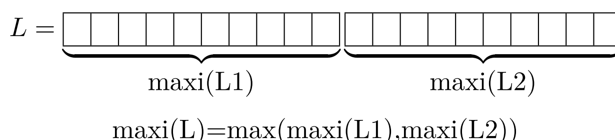
- **Diviser :** c'est découper le problème en sous-problèmes identiques mais sur des entrées de tailles inférieures.
- **Régner :** c'est pouvoir résoudre tous les sous-problèmes soit directement (cas de base), soit de manière récursive.
- **Combiner :** c'est résoudre le problème initial à partir des solutions des sous-problèmes.

On applique cette méthode sur deux problèmes simples ayant trait aux tableaux (ou listes).

3.1 Problème 1 : le maximum récursif

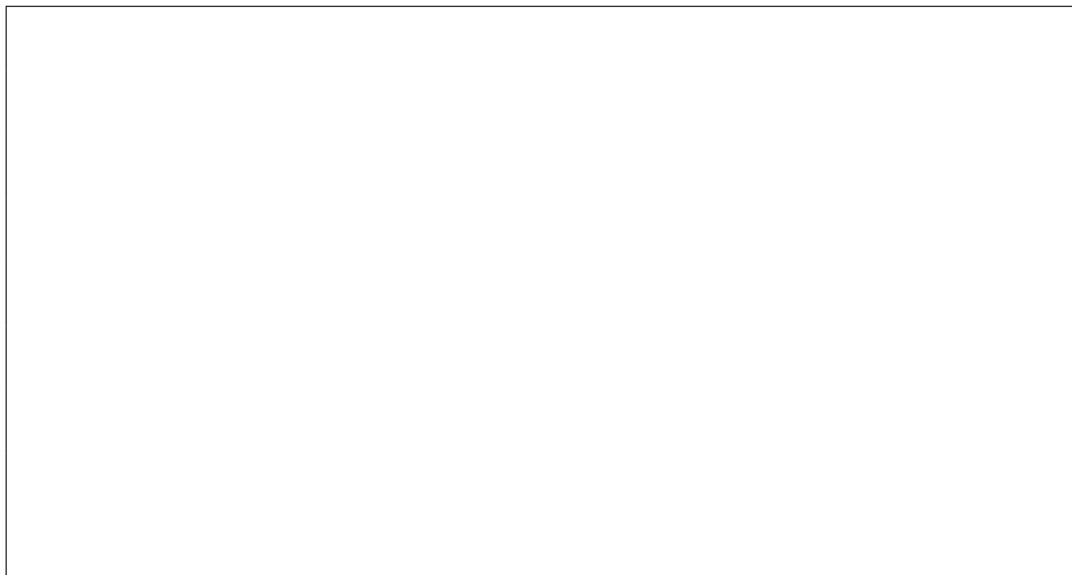
On cherche à résoudre le problème de recherche du maximum d'une liste `L`. La stratégie du *diviser pour régner* suggère l'approche suivante :

- **Diviser :** on découpe la liste en deux sous-listes de taille égale (ou presque) pour lesquelles on se pose le problème du maximum ;
- **Régner :** si une sous-liste ne contient qu'un seul élément son maximum est cet élément, sinon on le trouve de manière récursive ;
- **Combiner :** ayant trouvé le maximum des deux sous-listes, on cherche la plus grande des deux valeurs.



maxi(L)=max(maxi(L1),maxi(L2))

Fonction récursive `maxi(L,i,j)` qui cherche le maximum de la liste `L` entre les indices `i` et `j`.

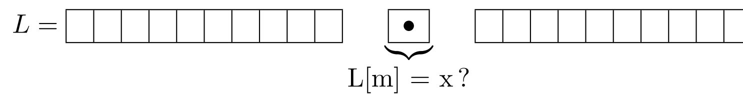


Analyse de la complexité de l'algorihtme (Nombre d'appels récursifs, nombre de comparaisons) : vérifier que la complexité est en $\mathcal{O}(n)$.

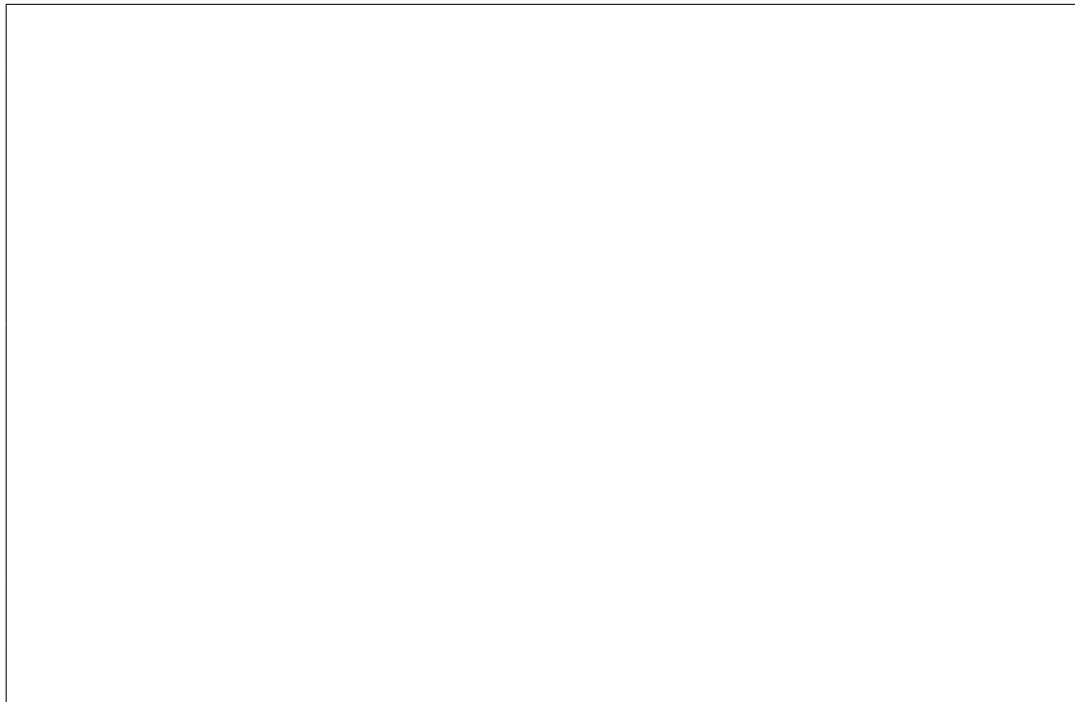
3.2 Problème 2 : recherche dichotomique récursive

On cherche à résoudre le problème de recherche d'un élément `x` dans une liste `L` d'objets triés par ordre croissant (par exemple : une liste de nombres, une liste de mot pour l'ordre alphabétique). On applique encore la stratégie du *diviser pour régner*

- **Diviser :** on se place au milieu (ou presque) de la liste (position d'indice m) ; on découpe alors la liste en deux sous-listes.
- **Régner :** si une sous-liste ne contient qu'un seul élément la réponse est obtenue en testant l'égalité avec `x`, sinon la réponse pour une sous-liste est obtenue de manière récursive.
- **Combiner :** on compare `L[m]` et `x` : s'ils sont égaux on a trouvé, si `L[m]>x` on cherche dans la sous-liste à gauche, sinon on cherche dans la sous-liste à gauche.



Fonction récursive `rechDicho(L,x,i,j)` qui cherche l'élément `x` dans liste triée `L` entre les indices `i` et `j`. La fonction retourne l'indice de `x` s'il est dans la liste et `False` sinon.



Analyse de la complexité de l'algorihtme (Nombre d'appels récursifs, nombre de tests d'égalité) : vérifier que la complexité est en $\mathcal{O}(\log_2(n))$.

4 Transformée de Fourier

La transformée de Fourier discrète est un outil majeur en traitement du signal et intervient dans des domaines très variés.

4.1 Un contexte : spectre d'un signal périodique

Un signal f, de période T, se développe en série de Fourier sous la forme :

$$f(t) = \sum_{n=-\infty}^{+\infty} c_n e^{2i\pi\, n \frac{t}{T}}$$

où c_n désignent les coefficients de Fourier qui se calculent sous la forme intégrale :

$$c_n = \frac{1}{T} \int_0^T f(t) e^{-2i\pi\, n \frac{t}{T}} \,\mathrm{d}\, t$$

Le spectre de f est l'ensemble des couples $\left(\frac{n}{T}, c_n\right)_{n\in\mathbb{Z}}$. La détermination du spectre est un élément important de l'étude d'un signal périodique.

On suppose connue la période T du signal ainsi qu'un nombre fini N de valeurs du signal régulièrement espacés sur une période :

$$y_k = f\left(k\frac{T}{N}\right) \text{ avec } k = 0, 1, 2 \cdots N-1$$

La méthode des rectangle nous permet d'approcher les coefficients de Fourier par la somme ci-dessous :

$$\frac{1}{N} \sum_{k=0}^{N-1} y_k e^{-2i\pi\, n \frac{k}{N}} \simeq c_n$$

On vérifie que l'approximation est essentiellement valable pour $-\frac{N}{2} \leq n \leq \frac{N}{2} - 1$ (ou un intervalle centré si N est impair). C'est sur le calcul de ces coefficients que se concentre notre étude.

4.2 Transformée de Fourier discrète

Définition. Nous fixons un entier $N > 0$ et nous posons $\omega_N = e^{\frac{2i\pi}{N}}$.
L'application qui a une famille $(y_k)_{0\leq k\leq N-1}$ de N nombres complexes associe la famille $(Y_n)_{0\leq n\leq N-1}$ définie par :

$$Y_n = \sum_{k=0}^{N-1} y_k \omega_N^{-nk}$$

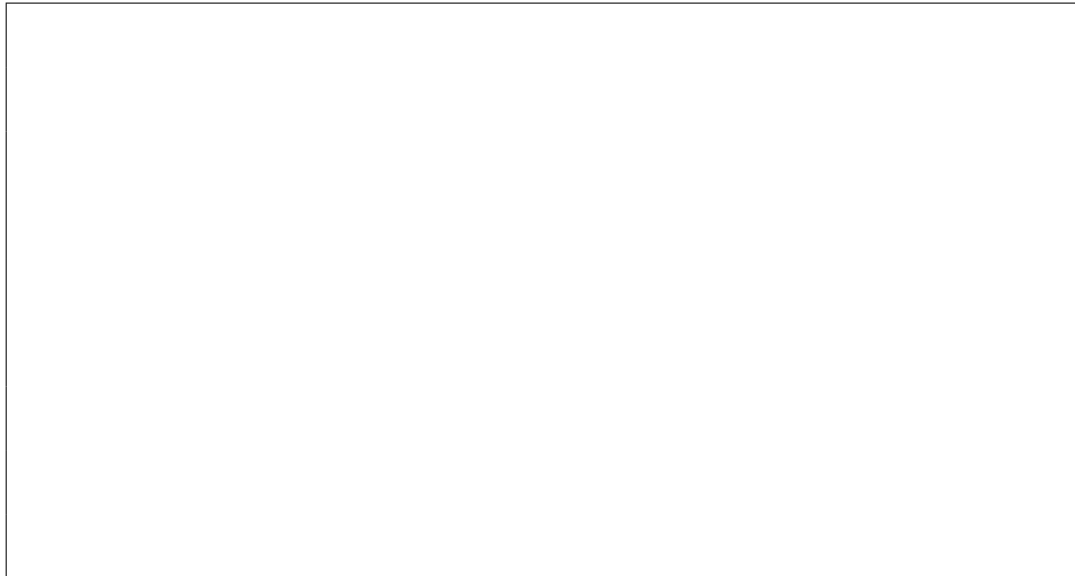
s'appelle la **transformée de Fourier discrète**.

Dans le calcul des coefficients de Fourier (cas N pair) nous avons donc :

$$N c_n \simeq \begin{cases} Y_n & \text{si } 0 \leq n < \frac{N}{2} \\ Y_{n+N} & \text{si } -\frac{N}{2} \leq n < 0 \end{cases}$$

Calcul direct.

1. Écrire une fonction Python `TFD(y)` qui prend une liste `y` en entrée et retourne la liste `Y` obtenue par transformée de Fourier discrète des coefficients de `y`. La somme est calculée de manière itérative.
2. Estimer le nombre d'additions et de multiplications nécessaires pour ce calcul.



Nombre d'opérations ?

Comment qualifier la complexité ?

4.3 Transformée de Fourier rapide

Nous présentons dans cette dernière partie un algorithme connu sous le nom de *transformée de Fourier rapide* (TFR) ou *fast Fourier transform* (FFT) qui va nettement faire baisser le coût de calcul de la tranformée de Fourier discrète.

La méthode : principe général. C'est une autre illustration du paradigme *diviser pour régner.* On suppose que N est un nombre pair, soit $N = 2m$.

1. Nous avons pour $0 \leq n < N$, $Y_n = P_n + \omega_N^{-n} I_n$ avec :

   $$P_n = y_0 + y_2\omega_N^{-2n} + \cdots + y_{2m-2}\omega_N^{-(2m-2)n} \quad \text{et} \quad I_n = y_1 + y_3\omega_N^{-2n} + \cdots + y_{2m-1}\omega_N^{-(2m-2)n}$$

   On remarque alors les relations : $P_{n+m} = P_n$ et $I_{n+m} = I_n$. Il suffit donc de calculer P_n et I_n pour $n = 0...m-1$ et remarquer que $\omega_N^{-(n+m)} = -\omega_N^{-n}$ pour former :

   $$Y_n = P_n + \omega_N^{-n} I_n$$

   $$\text{et} \quad Y_{n+m} = P_n + \omega_N^{-(n+m)} I_n = P_n - \omega_N^{-n} I_n$$

2. En se rappelant que $\omega_N^2 = \omega_m$, une lecture attentive des nombres P_n et I_n nous permet d'écrire :

   $$(P_n)_{0 \leq n < m-1} = \text{TFD}(y_0, y_2, \cdots, y_{2m-2}) \quad \text{et} \quad (I_n)_{0 \leq n < m-1} = \text{TFD}(y_1, y_3, \cdots, y_{2m-1}).$$

L'algorithme. On décrit le principe d'une fonction `FFT(y,N)` qui prend en entrée une liste `y` de taille `N` et retourne la transformée de Fourier discrète de `y`. On suppose que N est une puissance de 2.

- **Diviser :** on découpe la liste en deux sous-listes, celle des termes de rang pair et celle des termes de rang impair.
- **Régner :** si une sous-liste ne contient qu'un seul élément le résultat est immédiat, sinon on le trouve de manière récursive ;
- **Combiner :** ayant trouvé la TFD des deux sous-listes, on combine les listes selon le point 1 de la méthode décrite ci-dessus.

Fonction `FFT(y,N)`.

Analyse de la complexité.