# TD d'algorithmique avancée

## TD2 : La récursivité

### Suite de Fibonacci

La suite de Fibonacci est définie comme suit :

$$Fib(n) = \begin{cases} 1 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ Fib(n-1) + Fib(n-2) & \text{sinon} \end{cases}$$

1. Ecrivez un algorithme récursif calculant *Fib*(***n***).
2. Calculez ça complexité en nombre d'additions.
3. Ecrire un algorithme pour une fonction récursive appelée FIB_N_FIB_N_1() qui calcule, pour ***n***>0, le couple (*Fib*(***n***), *Fib*(***n***-1)).
4. Utilisez la fonction FIB_N_FIB_N_1() pour écrire un nouvel algorithme calculant *Fib*(***n***).
5. Calculez ça complexité en nombre d'additions.

### Opérations sur les ensembles

1. Soit une fonction qui vérifie si un élément '*x*' appartient à une partie d'un tableau 'A' appelée IS_MEMBER() qui reçoit en paramètres le tableau A, l'élément '*x*' et les indices de la partie du tableau 'd' et 'f' et retourne 'true' ou 'false'.
    a. Si l'ensemble 'A' n'est pas trié :
        i. Ecrivez un algorithme récursif pour cette fonction ;
        ii. Calculez sa complexité en nombre de comparaisons avec les éléments du tableau.
    b. Si l'ensemble 'A' est trié dans l'ordre croissant :
        i. Ecrivez un algorithme récursif pour cette fonction ;
        ii. Calculez sa complexité en nombre de comparaisons avec les éléments du tableau ;
        iii. Utilisez la recherche dichotomique pour améliorer votre algorithme ;
        iv. Calculez la complexité du nouvel algorithme en nombre de comparaisons avec les éléments du tableau.
2. Soit les fonctions :

    UNION(A,B) qui fait l'union de deux ensembles ;

    INTERSECT(A,B) qui fait l'intersection de deux ensembles ;

    a. Si les ensembles 'A' et 'B' ne sont pas triés :
        i. Ecrivez des algorithmes récursifs pour ces fonctions ;
        ii. Calculez leurs complexités en nombre de comparaisons avec les éléments des tableaux.
    b. Si les ensembles 'A' et 'B' sont triés dans l'ordre croissant :
        i. Ecrivez des algorithmes récursifs pour ces fonctions ;
        ii. Calculez leurs complexités en nombre de comparaisons avec les éléments des tableaux.

# TD d'algorithmique avancée

## Corrigé du TD2 : La récursivité

### Suite de Fibonacci

La suite de Fibonacci est définie comme suit :

$$Fib(n) = \begin{cases} 1 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ Fib(n-1) + Fib(n-2) & \text{sinon} \end{cases}$$

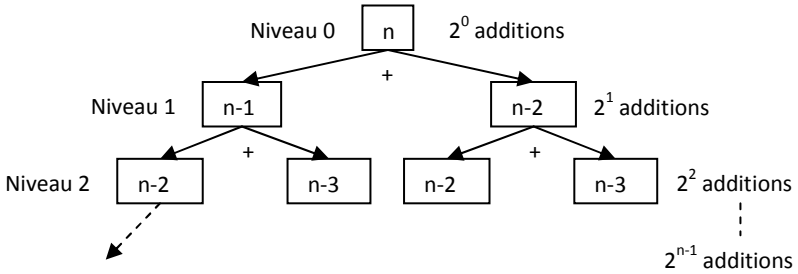
1. Ecrivez un algorithme récursif calculant *Fib*(***n***).

```
FIB(n)
   si n=0 ou n=1 alors retourner 1
   sinon retourner FIB(n-1) + FIB(n-2)
```

(Récursivité multiple.)

2. Calculez ça complexité en nombre d'additions.

L'algorithme s'exécute sous forme d'un arbre binaire comme suit :




Nombre total d'additions au maximum = $2^0+2^1+2^2+\ldots+2^{n-1}$ = somme d'une suite géométrique = $u_0 \cdot \frac{1-b^{\text{nombre d'éléments}}}{1-b} = 1 \cdot \frac{1-2^n}{1-2} = 2^n - 1$

La complexité est de l'ordre de : $O(2^n) = 2^n - 1$ (Complexité exponentielle)

3. Ecrire un algorithme pour une fonction récursive appelée FIB_N_FIB_N_1() qui calcule, pour ***n***>0, le couple (*Fib*(***n***), *Fib*(***n***-1)).

```
FIB_N_FIB_N_1(n)
   si n=1 alors retourner(1, 1)
   sinon (x, y) = FIB_N_FIB_N_1(n-1)
   retourner(x+y, x)
```

4. Utilisez la fonction FIB_N_FIB_N_1() pour écrire un nouvel algorithme calculant *Fib*(***n***).

```
FIB(n)
   si n=0 alors retourner 1
   sinon (x,y) = FIB_N_FIB_N_1(n)
   retourner x
```

5. Calculez ça complexité en nombre d'additions.

O(n)=n-1

### Opérations sur les ensembles

1. Soit une fonction qui vérifie si un élément '*x*' appartient à une partie d'un tableau 'A' appelée IS_MEMBER() qui reçoit en paramètres le tableau A, l'élément '*x*' et les indices de la partie du tableau 'd' et 'f' et retourne 'true' ou 'false'.

    a. Si l'ensemble 'A' n'est pas trié :

        i. Ecrivez un algorithme récursif pour cette fonction ;

```
IS_MEMBER(A, x, d, f)
   si d>f alors retourner False
   si A[d]=x alors retourner True
   sinon retourner IS_MEMBER(A, x, d+1, f)
```

        ii. Calculez sa complexité en nombre de comparaisons avec les éléments du tableau.

        Au cas pire, '*x*' n'appartient pas au tableau : O(**f-d**) = **f-d+1**

    b. Si l'ensemble 'A' est trié dans l'ordre croissant :

        i. Ecrivez un algorithme récursif pour cette fonction ;

```
IS_MEMBER(A, x, d, f)
   si d>f ou A[d]>x alors retourner False
   si A[d]=x alors retourner True
   sinon retourner IS_MEMBER(A, x, d+1, f)
```

        ii. Calculez sa complexité en nombre de comparaisons avec les éléments du tableau ;

        Au cas pire, '*x*' n'appartient pas au tableau et il est plus grand que tous les éléments : O(**f-d**) = 2(**f-d+1**)

        iii. Utilisez la recherche dichotomique pour améliorer votre algorithme ;

```
IS_MEMBER(A, x, d, f)
   si d=f alors      si A[d]=x alors retourner True
                     sinon retourner False
   m←[(d+f)/2]
   si A[m]=x alors retourner True
   sinon si A[m]>x alors retourner IS_MEMBER(A, x, d, m-1)
         sinon retourner IS_MEMBER(A, x, m+1, f)
```

        iv. Calculez la complexité du nouvel algorithme en nombre de comparaisons avec les éléments du tableau.

        Au cas pire où '*x*' n'appartient pas au tableau : $O(\log_2(\mathbf{f\text{-}d}))=2*\log_2(\mathbf{f\text{-}d+1})+1$

        [ log2(**f-d+1**) appels à la fonction dont chaque appel contient 2 comparaisons + une comparaison quand d=f ]

2. Soit les fonctions :

    UNION(A,B) qui fait l'union de deux ensembles ;

    INTERSECT(A,B) qui fait l'intersection de deux ensembles ;

    a. Si les ensembles 'A' et 'B' ne sont pas triés :

        i. Ecrivez des algorithmes récursifs pour ces fonctions ;

```
UNION(A, B, idxB, C)
   Si (taille(C)=0) alors
         pour i=1 à taille(A) faire
               taille(C) ← taille(C)+1
               C[i] ← A[i]
   si (IS_MEMBER(A, B[idxB], 1, taille(A)) = False)
   alors taille(C) ← taille(C)+1
         C[taille(C)] ← B[idxB]
   si (idxB < taille(B)) alors UNION(A, B, idxB+1, C)
```

L'appel initial est :

```
UNION(A, B, 1, C)
```

```
INTERSECT(A, B, idxB, C)
   si (IS_MEMBER(A, B[idxB], 1, taille(A)) = True)
   alors taille(C) ← taille(C)+1
         C[taille(C)] ← B[idxB]
   si (idxB < taille(B)) INTERSECT(A, B, idxB+1, C)
```

L'appel initial est :

```
INTERSECT(A, B, 1, C)
```

ii. Calculez leurs complexités en nombre de comparaisons avec les éléments des tableaux.

UNION : $O(m.\log_2(n)) = m*(2*\log_2(n)+1)$, avec $m$ taille de B et $n$ taille de A.

INTERSECT : $O(m.\log_2(n)) = m*(2*\log_2(n)+1)$, avec $m$ taille de B et $n$ taille de A.

b. Si les ensembles 'A' et 'B' sont triés dans l'ordre croissant :

i. Ecrivez des algorithmes récursifs pour ces fonctions ;

```
UNION(A, idxA, B, idxB, C)
   si (idxA ≤ taille(A) ou idxB ≤ taille(B)) alors
   ⌈      si (idxB > taille(B) ou B[idxB] > A[idxA]) alors
   |      ⌈      taille(C) ← taille(C) + 1
   |      |      C[taille(C)] ← A[idxA]
   |      ⌊      UNION(A, idxA+1, B, idxB, C)
   |      sinon
   |      ⌈      taille(C) ← taille(C) + 1
   |      |      C[taille(C)] ← B[idxB]
   |      |      si (idxA<=taille(A) et A[idxA]=B[idxB]) alors UNION(A,idxA+1,B,idxB+1,C)
   ⌊      ⌊      sinon UNION(A, idxA, B, idxA+1, C)
```

L'appel initial est :

```
UNION(A, 1, B, 1, C)
```

```
INTERSECT(A, idxA, B, idxB, C)
   si (idxA ≤ taille(A) et idxB ≤ taille(B)) alors
   ⌈      si (A[idxA] = B[idxB]) alors
   |      ⌈      taille(C) ← taille(C)+1
   |      |      C[taille(C)] ← A[idxA]
   |      ⌊      INTERSECT(A, idxA+1, B, idxB+1, C)
```

```
|       sinon ⌈si (B[idxB] > A[idxA]) alors INTERSECT(A, idxA+1, B, idxB, C)
⌊             ⌊sinon INTERSECT(A, idxA, B, idxB+1, C)
```

L'appel initial est :

```
INTERSECT(A, 1, B, 1, C)
```

ii. Calculez leurs complexités en nombre de comparaisons avec les éléments des tableaux.

UNION : Pour chaque case de C on appel UNION() une fois. Et chaque appel de UNION() fait au max 2 comparaisons. Et la taille de C au max est m+n. Donc O(***m+n***) = 2.(***m+n***), avec ***m*** taille de B et ***n*** taille de A.

INTERSECT : Le cas pire est quand tous les éléments du petit tableau se trouvent dans le grand, et la dernière case du grand tableau se trouve dans le petit pour parcourir tout le grand tableau. Ce qui fait qu'on appel INTERSECT() autant de fois qu'il y a de case dans le grand tableau. Et chaque appel fait au max 2 comparaisons. Donc O(***n***) = 2*****n***, avec ***n*** la taille du grand tableau.