I. Notions fondamentales

Gérard Berry (collège de France) donne la définition suivante d'algorithme :

> *Un algorithme, c'est tout simplement une façon de décrire dans ses moindres détails comment procéder pour faire quelque chose.*

Un algorithme ne dépend d'aucun formalisme et est indépendant de tout langage de programmation. Nous rencontrons de nombreux algorithmes dans la vie de tous les jours :

— une recette de cuisine : des instructions permettant de réaliser un plat,
— un trajet routier : des instructions permettant d'atteindre une destination,
— poser et résoudre une multiplication comme en CE2 : une méthode permettant de calculer le résultat d'une multiplication.

Un programme quant à lui est une représentation en mémoire d'une suite d'instruction exécutables pas un processeur. Il s'agit de l'écriture d'une collection d'algorithmes dans un langage qui puisse être interprété par une machine.

Voici par exemple un algorithme retournant les racines du polynôme $a\,x^2 + b\,x + c$ avec $a \neq 0$.

```
fonction RACINES(a, b, c)
    d ← b² − 4 a c
    si d < 0 alors
        retourne ∅
    sinon si d = 0 alors
        retourne {−b/(2 a)}
    sinon
        retourne {(−b−√d)/(2 a), (−b+√d)/(2 a)}
    fin si
fin fonction
```

1. Algorithme itératif

Supposons que l'on souhaite calculer la somme w de deux vecteurs u et v de $\mathbb{R}^{100}$. Mathématiquement, nous écrivons :

> Pour $u \in \mathbb{R}^{100}$ et $v \in \mathbb{R}^{100}$, le vecteur $w = u + v$ est donné par $w_1 = u_1 + v_1$, ..., $w_{100} = u_{100} + v_{100}$.

Nous obtenons alors l'algorithme suivant.

```
fonction SOMME(u, v)
    Création d'un vecteur w de ℝ¹⁰⁰
    w₁ ← u₁ + v₁
    ...
    retourne w
fin fonction
```

Nous pouvons continuer ainsi l'écriture de l'algorithme mais c'est un peu répétitif.

Maintenant supposons que la dimension du vecteur n'est pas fixé à 100 mais soit un paramètre $n \in \mathbb{N}$. Nous aimerions disposer d'un algorithme SOMME_N(u, v, n) retournant la somme de deux vecteurs u et v de $\mathbb{R}^n$ avec $n \in \mathbb{N}$. Du point de vu mathématique il n'y a pas de problème :

> Pour $n \in \mathbb{N}$, $u \in \mathbb{R}^n$ et $v \in \mathbb{R}^n$, le vecteur $w = u + v$ est donné par $w_1 = u_1 + v_1$, ..., $w_n = u_n + v_n$.

Comment préciser dans l'algorithme que nous devons effectuer n tâches similaires ? C'est pour résoudre ces problèmes de répétition de tâches que nous utilisons l'*itération*.

```
fonction SOMME_N(u, v, n)
    Création d'un vecteur w de ℝ^n
    pour i de 1 à n faire
        w_i = u_i + v_i
    fin pour
    retourne w
fin fonction
```

La boucle **pour** ne suffit pas toujours. Ecrivons un algorithme VALUATION qui étant un entier quelconque $n \in \mathbb{N} \setminus \{0\}$ et un premier p retourne le plus grand entier $k \in \mathbb{N}$ tel que p^k divise n. Nous commençons par poser $i = 1$. Si p^i divise n, on recommence en posant $i = i + 1$ sinon on retourne $i - 1$. En fait nous incrémentons la valeur de i tant que p^i divise n.

```
fonction VALUATION(p, n)
    i ← 1
    tant que p^i divise n faire
        i ← i + 1
    fin tant que
    retourne i − 1
fin fonction
```

Montrons que cet algorithme est correct. Soit $n \in \mathbb{N}$ et p un nombre premier. Posons

$$X = \{i \in \mathbb{N} \text{ tel que } p^i \text{ ne divise pas } n\}.$$

L'ensemble $X \subseteq \mathbb{N}$ est non vide, il contient donc un plus petit élément $m \in \mathbb{N}$. Comme $p^0 = 1$ est un diviseur de n, nous avons $m \geq 1$. Dans l'algorithme nous quittons la boucle **tant que** lorsque i vaut m. Les valeurs de k tels que p^k divise n sont donc $\{0, \ldots, m-1\}$ et l'entier recherché est $m - 1$.

Remarquons que nous n'aurions pas pu utiliser une boucle **pour** pour cet algorithme car nous ne connaissons pas à l'avance le nombre de "pas de boucle" nécessaires. C'est pour cela que nous avons utiliser une boucle **tant que**.

Exercice. Réécrire l'algorithme SOMME_N à l'aide d'une boucle **tant que**.

Correction.

```
fonction SOMME_N(u, v, n)
    Création d'un vecteur w de ℝ^n
    i ← 1
    tant que i ≤ n faire
        w_i = u_i + v_i
        i ← i + 1
    fin tant que
    retourne w
fin fonction
```

Comme toute boucle **pour** peut-être remplacée par une boucle **tant que** pour quoi les utilise t-on encore ? Tout simplement car nous somme *a priori* pas certain qu'une boucle **tant que** se finisse, ce qui peut être source de *bug*.

Exercice. Considérons l'algorithme suivant prenant en paramètre un entier $k \in \mathbb{Z}$.

```
fonction ALGO(k)
    t ← k
    i ← 0
    tant que t ≠ 0 faire
        i ← i + 1
        t ← t − 2
    fin tant que
    retourne i
fin fonction
```

1. A quelle(s) condition(s) sur k l'algorithme ALGO termine t-il ?

2. S'il termine que vaut l'entier retourné ?

Correction.

1. L'entier k doit être positif et pair.

2. Si k est positif et pair alors l'entier retouré est $k/2$.

2. Démonstration par récurrence

Revenons un court instant à la démonstration par récurrence. Soit $\mathcal{P}(n)$ une propriété dépendante d'un entier $n \in \mathbb{N}$. Par exemple, nous pouvons considérer

$$\mathcal{P}(n) : \quad \sum_{k=0}^{n} k = \frac{n(n+1)}{2}.$$

Au Lycée vous avez appris que pour démontrer $\mathcal{P}(n)$ pour tout $n \geq n_0$, il suffisait de démontrer

– **initialisation** : $\mathcal{P}(n_0)$,

– **hérédité** : pour tout $n \geq n_0$, $\mathcal{P}(n) \Rightarrow \mathcal{P}(n+1)$.

L'entier n_0 est alors appelé *base* de la récurrence.

Mais pourquoi la démonstration par récurrence est-elle valide ? Supposons que nous ayons établi l'initialisation et l'hérédité. De plus supposons par l'absurde que $\mathcal{P}(N)$ soit faux pour un certains $N \geq n_0$. Posons

$$X = \{n \in \mathbb{N}, \text{ tel que } n \geq n_0 \text{ et } \mathcal{P}(n) \text{ soit faux}\}.$$

Par hypothèse $N \in X$ et donc X est non vide et contient un plus petit élément $m \geq n_0$, en particulier $P(m)$ est faux. Par **initialisation** $n_0 \notin X$ et donc $m > n_0$. Comme $m-1$ vérifie $m-1 \geq n_0$ et $m-1 \notin X$, la propriété $\mathcal{P}(m-1)$ est vrai et donc aussi $\mathcal{P}(m)$ par **hérédité**. Contradiction. Et donc $\mathcal{P}(n)$ est vraie pour tout $n \geq n_0$.

3. Algorithme récursif

Certains objets mathématiques ou informatiques peuvent être définis de façon récursive. Prenons l'exemple de la suite de Fibonacci $(F_n)_n$. Elle est définie de façon récursive par $F_0 = 0$, $F_1 = 1$ et $F_n = F_{n-1} + F_{n-2}$ pour tout $n \geq 2$. Les égalités $F_0 = 0$ et $F_1 = 1$ forment la base de cette définition récursive.

Pour qu'une définition récursive soit correcte il faut absolument qu'il y ait une base et que la définition d'un objet hors de la base fasse appel à des objets "plus petits" menant pas à pas vers un objet de la base Voici, par exemple une définition possible de "mot" en informatique :

– le mot vide $\emptyset$ est un mot
– une lettre suivi d'un mot est un mot

Nous avons bien que ab est un mot. En effet ab est la lettre a suivie de b qui est un mot car c'est la lettre b suivie du mot vide $\emptyset$. Pour définir un mot de longueur ℓ, nous utilisons donc les mots de longueur $\ell - 1$ et au final nous retombons sur le mot vide qui est de longueur 0.

Tout comme une définition récursive, un algorithme récursif est un algorithme qui s'appelle lui-même. Sa construction est très semblable à ce que nous venons de voir. En particulier, il est important de bien identifier la base de la récursion.

Voici un algorithme récursif retournant le n-ème terme F_n de la suite de Fibonacci pour $n \in \mathbb{N}$.

```
fonction FIBO_REC(n)
    si n = 0 alors
        retourne 0
    sinon si n = 1 alors
        retourne 1
    sinon
        retourne FIBO_REC(n − 1) + FIBO_REC(n − 2)
    fin si
fin fonction
```

4. Compléxité

Deux algorithmes distincts peuvent effectuer la même tâche. Par exemple l'algorithme suivant retourne aussi le n ème terme F_n de la suite de Fibonacci pour tout $n \in \mathbb{N}$.

```
1:  fonction FIBO_ITER(n)
2:      si n = 0 alors
3:          retourne 0.
4:      sinon
5:          u_p ← 0
6:          u ← 1
7:          pour i de 2 à n faire
8:              t ← u
9:              u ← u + u_p
10:             u_p ← t
11:         fin pour
12:         retourne u
13:     fin si
14: fin fonction
```

Exercice. Montrons que l'algorithme évalué en $n \in \mathbb{N}$ retourne bien le n-ème terme F_n de la suite de Fibonacci.

1. Vérifier que c'est le cas pour $n = 0, 1$ et 2.

2. A quels termes de la suite de Fibonacci correspondent les variables u et u_p après exécution de la ligne 6.

3. Montrer par récurrence sur $i \in \{2, \dots, n\}$ qu'après exécution de la ligne 10 nous avons $u = F_n$ et $u_p = F_{n-1}$.

4. Conclure.

Correction.

1. Pour $n = 0$, l'algorithme retourne 0 qui est la valeur de F_0. Pour $n = 1$, l'algorithme exécute les lignes 5 et 6 et retourne le contenu de u qui vaut 1 et qui correspond à F_1. Pour $n = 2$, après la ligne 6 nous avons $u = 1$ et $u_p = 0$. Après les lignes 8, 9 et 10 nous avons respectivement $t = 1$, $u = 1 + 0 = 1$ et $u_p = 1$ et nous quittons la boucle **pour**. La valeur retournée est celle contenue dans u, à savoir 1, qui est aussi la valeur de F_2.
2. Après exécution de la ligne 6 nous avons $u_p = F_0$ et $u = F_1$.
3. Notons H_j l'hypothèse de récurence « Après exécution de la ligne 10 pour le pas de boucle $i = j$, on a $u = F_j$ et $u_p = F_{j-1}$ » . Ce que nous avons fais au **1** pour $n = 2$ établit H_2. Supposons H_j pour $j \in [2, n-1]$ et montrons H_{j+1}. Par hypothèse de récurrence avant d'exécuter la ligne 8 pour la boucle $i = j$ nous avons $u = F_j$ et $u_p = F_{j-1}$. Après exécution des lignes 8, 9 et 10 nous avons respectivement $t = F_j$, $u = F_j + F_{j-1} = F_{j+1}$ et $u_p = F_j$, ce qui montre H_{j+1}.
4. Supposons $n \geq 2$. D'après ce qui précéde, après exécution de la boucle pour $i = n$ nous avons $u = F_n$ et $u_p = F_{n-1}$ et donc l'algorithme retourne F_n. Par le **1** ce résultat est encore valide pour $n = 0$ et $n = 1$.

Parmi les algorithmes FIBO_REC et FIBO_ITER lequel choisir ? Selon quel critère ? La première version est plus facile à comprendre que la seconde mais ce critère n'est pas valable d'un point de vue programmation. Un meilleur critère est le temps d'exécution.

Pour le moment nous allons nous limiter à compter le nombre d'additions. Notons a_n, *resp* b_n le nombre d'additions effectuées lors de l'appel de FIBO_REC(n), *resp* FIBO_ITER(n). Nous avons $a_0 = a_1 = 0$ et $a_n = 1 + a_{n-1} + a_{n-2}$ pour $n \geq 2$ ainsi que $b_0 = b_1 = 0$ et $b_n = b_{n-1} + 1$, ce qui donne le tableau suivant :

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
a_n	0	0	1	2	4	7	12	20	33	54	88	143	232	376	609	986	1596
b_n	0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Exercice. Montrer que pour tout $n \geq 1$ nous avons $a_n = F_{n+1} - 1$ et $b_n = n - 1$.

Correction. Comme la boucle **pour** de l'algorithme FIBO_ITER est de taille $n - 1$ et qu'exactement une addition est effectuée par pas de boucle, nous avons bien $b_n = n - 1$ pour $n \geq 2$. On conclut en remarquant $b_1 = 0$. Traitons maintenant le cas de a_n. Une analyse rapide donne $a_0 = 0$, $a_1 = 0$ et $a_n = 1 + a_{n-1} + a_{n-2}$ pour $n \geq 2$. Montrons par récurrence sur $n \geq 0$ que nous avons $a_n = F_{n+1} - 1$. C'est vrai pour $n = 0$ et $n = 1$ car nous avons $F_1 - 1 = 1 - 1 = 0$ et $F_2 - 1 = 1 - 1 = 0$. Supposons le résultat pour $n \geq 2$ et montrons le pour $n + 1$. Nous avons

$$a_{n+1} = 1 + a_n + a_{n-1} = 1 + F_{n+1} - 1 + F_n - 1 = F_{n+1} + F_n - 1 = F_{n+2} - 1.$$

Nous obtenons ainsi $a_n = F_{n+1} - 1$ pour tout $n \geq 0$.

Nous admettons l'équivalent $F_n \sim \frac{1}{\sqrt{5}}\varphi^n$ où $\varphi = \frac{1+\sqrt{5}}{2}$.

Rappel : $u_n \sim v_n$ si $v_n \neq 0$ pour tout $n \in \mathbb{N}$ et $\lim\limits_{n \to +\infty} \dfrac{u_n}{v_n} = 1$.

Définition 1. Soient f et g deux fonctions de $\mathbb{N}$ dans $\mathbb{R}$. Nous disons que $f(n)$ est en $O(g(n))$ s'il existe un entier $N \in \mathbb{N}$ et un réel $C \in \mathbb{R}$ tel que $|f(n)| \leq C|g(n)|$ pour tout $n \geq N$.

Dans ce cours $g(n)$ sera souvent une combinaison de $\log(n)$, n^k et c^n pour $k \in \mathbb{N}$ et $c \in \mathbb{R}$. Remarquons que si $f(n)$ est en $O(g(n))$ alors $\lambda f(n)$ est aussi en $O(g(n))$ pour tout $\lambda \in \mathbb{R}^*$.

La suite a_n est $O(\varphi^n)$ tandis que la suite b_n est en $O(n)$. Nous disons que la croissance de a_n est exponentielle tandis que celle de b_n est linéaire. Comme φ^n domine n, l'algorithme FIBO_ITER est assymptotiquement plus rapide que FIBO_REC.

Essayons de prendre conscience de ce phénomène. Supposons qu'un ordinateur fasse 1 milliard d'additions par seconde (ce qui est réaliste). Le calcul de F_{140} à l'aide de FIBO_ITER prendra moins d'une millionième de seconde tandis que le même calcul utilisant FIBO_REC prendra 10^{18} secondes, soit plus de 2 fois l'âge de l'univers.