I – Introduction

De nombreux langages de programmation (des centaines...) existent ! Bien entendu, certains présentent des similarités, mais des fonctionnalités bien différentes peuvent aussi apparaître. Il existe notamment différents **paradigmes** de programmation qui correspondent à la façon d'envisager et de concevoir un programme (ceci sera développé plus précisément en classe de Terminale).

Au final, un langage de programmation finit toujours par être traduit en langage machine qui dépend de l'architecture matérielle sur laquelle le programme doit s'exécuter. Les langages de haut niveaux qui suivent le paradigme de programmation **impératif** (et procédural) se ressemblent assez et sont en quelque sorte une réécriture plus simple pour les humains du langage machine. Les langages qui suivent le paradigme **fonctionnel** s'écartent de l'aspect impératif du langage machine (suite d'instructions avec un aspect séquentiel) pour décrire *ce que sont* les choses plutôt que de décrire *comment* calculer les choses.

Pour approfondir cette découverte de la diversité langages, vous pouvez consulter très utilement le site suivant : `https://www.literateprograms.org/`

II – Algorithme d'Euclide de pgcd en langage impératif

L'algorithme d'Euclide est une méthode efficace pour calculer le plus grand commun diviseur de 2 nombres entiers et fut l'un des premiers algorithmes à être décrit formellement.

Il repose sur les 2 identités suivantes :

- a > b implique : pgcd(a, b) = pgcd(b, a modulo b)
- pgcd(a, 0) = a

1. Implémentation en Python

```python
def pgcd(a, b):
    while b != 0:
        a, b = b, a%b
    return a
```

2. Implémentation en C

```c
int pgcd(int a, int b) {
    // il faut permuter a et b si b est plus grand que a
    if (b > a) {
        int temp = a;
        a = b;
        b = temp;
    }
    while (b != 0) {
        int m = a%b;
        a = b;
        b = m;
    }
    return a;
}
```

3. Conclusion

Les structures de ces langages sont très similaires. Voici quelques différences principales :

- Les variables doivent être déclarées, et typées. Ceci est une différence fondamentale et très importante avec Python.
- Les blocs de code sont délimités par des accolades (et non pas avec des indentations comme en Python, même si on tend à les utiliser pour facilité la lecture du code).
- Les autres différences ne relèvent que de la syntaxe précise du langage, mais n'ont rien de fondamental.

III – Algorithme d'Euclide de pgcd en langage fonctionnel

Rappel : les langages fonctionnels reposent beaucoup sur la notion de récursivité (qui sera étudiée en détail en classe de Terminale) : càd qu'une fonction peut s'appeler elle-même.

1. Implémentation en Erlang

```
pgcd(A, 0) -> A;
pgcd(A, B) -> pgcd(B, A rem B).
```

2. Implémentation en Haskell

```
pgcd a 0 = a
pgcd a b = pgcd b (a `mod` b)
```

3. Implémentation en OCaml

```
let rec pgcd a b =
    match b with
    | 0 -> a
    | b -> pgcd b (a mod b)
```

IV – Tri pas sélection en langage impératif

Rappel : le tri par sélection repose sur l'idée qu'au cours de l'algorithme, la partie gauche du tableau est triée, et qu'il reste la partie droite à trier. On doit donc rechercher le plus petit élément non trié et l'échanger avec le premier élément non trié. On progresse ainsi jusqu'à ce que la partie non triée soit vide.

1. Implémentation en Python

```
def triSelection(tab):
    for i in range(len(tab)-1):
        # recherche du minimum :
        i_mini = i
        for j in range(i+1, len(tab)):
            if tab[j] < tab[i_mini]:
                i_mini = j
        # permutation de tab[i] et minimum :
        tab[i], tab[i_mini] = tab[i_mini], tab[i]
```

2. Implémentation en C

```c
void triSelection(int tab[], int N) { // N est la taille du tableau
    int i;
    for(i=0; i<N; i++) {
        // recherche du minimum :
        int i_mini = i;
        int mini = tab[i];
        int j;
        for(j=i+1; j<N; j++) {
            if (tab[j]<mini) {
                i_mini = j;
                mini = tab[j];
            }
        }
        // permutation de tab[i] et minimum :
        tab[i_mini] = tab[i];
        tab[i] = mini;
    }
}
```

V – Tri pas sélection dans un langage fonctionnel (OCaml)

On découpe le programme en plusieurs fonctions modulaires.

1. **Calcul du minimum d'une liste :**
   L'idée globale est la suivante. Pour trouver le minimum d'une liste, on considère 2 cas :
   - D'abord un cas particulier où la liste ne contient qu'un seul élément *e*. Alors le minimum est forcément cet unique élément *e*.
   - Pour le cas d'une liste à plusieurs éléments, on décrit la liste suivant un **motif** `tête+queue`, où `tête` est le 1er élément de la liste et `queue` est la liste constituée de tous les autres éléments suivants.
     Le minimum de la liste est alors soit :
     a) l'élément `tête` s'il est inférieur au minimum de la `queue`,
     b) ou alors le minimum de la `queue` dans le cas contraire.

```
let rec minimum liste =
  match liste with
  | [e] -> e
  | tete :: queue ->
     let mini_queue = minimum queue in
         if tete < mini_queue then tete
         else mini_queue
```

   Quelques explications :
   - La 1ère ligne définit (*let*) une fonction `minimum` récursive (*rec*) qui prend un paramètre `liste`.
   - La 2ème ligne est une recherche de motif sur le paramètre *liste* (pattern matching).
     1) si la liste contient un seul élément *e*, la fonction vaut cet unique élément *e*.
     2) si la liste correspond au motif `tete::queue`, on définit (récursivement) le minimum de la queue, et en fonction de la comparaison de ce minimum avec la tête, la fonction vaut la tête ou ce minium.

Remarque : pour simplifier, on n'a pas traité le cas d'une liste vide.

2. **Suppression d'un élément d'une liste :**
Cette fonction prend un élément `e` et une liste en entrée et renvoie une liste qui ne contient plus cet élément (sa 1ère occurrence) :

```
let rec supprime e liste =
  match liste with
  | [] -> []
  | tete :: queue -> if e = tete then queue
                     else tete :: supprime e queue
```

Quelques explications :

- La 1ère ligne définit (*let*) une fonction `supprime` récursive (*rec*) qui prend 2 paramètres `e` et `liste`.
- La 2ème ligne est une recherche de motif sur le paramètre `liste` :
  1) si la liste est vide, la fonction vaut une liste vide.
  2) si la liste correspond au motif `tete::queue` :
  a) la fonction vaut la liste `queue` si l'élément `e` à supprimer est égal à la `tete`,
  b) sinon elle vaut une liste constituée la `tete` concaténée à la liste `queue` de laquelle on aura supprimé l'élément `e` (récursivement).

3. **Algorithme de tri complet :**
Cette fonction implémente le tri par sélection de façon complète :

```
let rec tri liste =
  match liste with
  | [] -> []
  | tete :: queue ->
    let mini = minimum liste in
      mini :: tri (supprime mini liste)
```

Quelques explications :

- La 1ère ligne définit une fonction `tri` récursive qui prend en paramètre la `liste` à trier.
- La 2ème ligne est une recherche de motif sur le paramètre `liste` :
  1) si la liste est vide, la fonction vaut une liste vide.
  2) si la liste correspond au motif `tete::queue` :
  a) on commence par définir le minimum de la liste `mini`,
  b) puis on indique que la la fonction vaut une liste constituée de `mini` concaténé à une liste obtenue en triant (récursivement) la `liste` de laquelle on aura supprimé l'élément minimum.

Exemple : avec la liste `[10; 3; 5; 4]` :

- `minimum liste` vaut 3 (`mini`)
- `supprime mini liste` vaut `[10; 5; 4]`
- `tri (supprime mini liste)` vaut `[4; 5; 10]`
- `mini :: tri (supprime mini liste)` vaut `[3; 4; 5; 10]`