Algorithmique et Programmation 2
Travaux Pratiques – Séance 2

À déposer à la fin de la présente séance sur Arche

> Avant de débuter cette séance de TP, assurez-vous d'avoir bien avancé, voire terminé, la feuille de la séance précédente.

Maintenant que vous maîtrisez la syntaxe du `C`, cette deuxième séance a pour but de vous entrainer à traduire les axiomes d'une fonction en un programme en `C`.

Pour chacun des exercices suivants, sauf mention contraire, chaque fonction en `C` demandée doit être accompagnée des procédures de test associées (voir exercice 1 du TP1).

Exercice 1 — Somme de nombres premiers

Soit la fonction `est_premier` qui à un entier naturel n associe vrai si n est premier et faux sinon (voir exercice 2 du TP1). On considère à présent la fonction `SPIA(n: entier naturel) : entier naturel` (pour *Somme des Premiers Inférieurs À*) qui à un entier naturel n associe la sommes des entiers naturels premiers et inférieurs ou égaux à n. Autrement dit,

$$\mathtt{SPIA}(n) = \sum_{\substack{p=1 \\ p \text{ premier}}}^{n} p$$

La fonction `SPIA(n: entier naturel) : entier naturel` satisfait les axiomes suivant : pour tout $n \in \mathbb{N}$,

[1] `SPIA(0) = SPIA(1) = 0`

[2] `SPIA(n+2) = (n+2) + SPIA(n+1)` si `est_premier(n+2)`

[3] `SPIA(n+2) = SPIA(n+1)` si non `est_premier(n+2)`

Écrire une fonction `C` pour `SPIA(n: entier naturel) : entier naturel` (nom du fichier à déposer sur Arche : `exo1.c`).

Exercice 2 — Suite et conjecture de Syracuse

En mathématiques, on appelle **suite de Syracuse** de l'entier $a > 0$ la suite $(u_n)_{n\in\mathbb{N}}$ d'entiers naturels définie comme suit, pour tout $n \in \mathbb{N}$:

$$\begin{aligned} u_0 &= a \\ u_{n+1} &= \begin{cases} \frac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{si } u_n \text{ est impair} \end{cases} \end{aligned}$$

Par exemple, la suite de Syracuse de l'entier 12 est 12, 6, 3, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1. . .

Soit la fonction `suivant(n: entier naturel) : entier naturel` qui à un entier naturel n associe le terme suivant dans une suite de Syracuse. La fonction `suivant` vérifie les axiomes suivants :

[1] `suivant(n) = n/2` si n est pair

[2] `suivant(n) = 3n+1` si n est impair

Soit `n_applique_suivant (n : entier naturel) : entier naturel` la fonction qui renvoie le nombre de fois qu'il faut appliquer la fonction `suivant`, en partant de n pour arriver à 1. Par exemple, pour $n = 12$, nous faisons appel neuf fois à la fonction `suivant` :

$$12 \xrightarrow{\texttt{suivant}} 6 \xrightarrow{\texttt{suivant}} 3 \xrightarrow{\texttt{suivant}} 10 \xrightarrow{\texttt{suivant}} 5 \xrightarrow{\texttt{suivant}} 16 \xrightarrow{\texttt{suivant}} 8 \xrightarrow{\texttt{suivant}} 4 \xrightarrow{\texttt{suivant}} 2 \xrightarrow{\texttt{suivant}} 1$$

Selon la conjecture de Syracuse [1], pour tout entier n > 0, `n_applique_suivant (n)` est bien défini. Autrement dit, toute suite de Syracuse finit par atteindre la valeur 1.

La fonction `n_applique_suivant (n : entier naturel) : entier naturel` vérifie les axiomes suivants :

[1] `n_applique_suivant(1) = 0`

[2] `n_applique_suivant(n) = 1 + n_applique_suivant(suivant(n))` si n > 1.

1. Écrire deux fonctions `C` pour `suivant(n: entier naturel) : entier naturel` et `n_applique_suivant (n : entier naturel) : entier naturel` (nom du fichier à déposer sur Arche : `exo2.c`).

1. https://fr.wikipedia.org/wiki/Conjecture_de_Syracuse

2. Modifier *a* minima la fonction `n_applique_suivant (n : entier naturel) : entier naturel` pour conserver en mémoire la plus grande valeur atteinte par la suite de Syracuse de n.

Exercice 3 — Fibonacci – Le retour !
Dans l'exercice 5 du TP1 (fichier `exo5.c`), vous avez programmé la fonction `fibonacci(n: entier naturel) : entier naturel` qui calcule puis retourne le terme de rang n de la suite de Fibonacci en utilisant la définition suivante :

$$f_n = \begin{cases} n & \text{si } n < 2 \\ f_{n-1} + f_{n-2} & \text{si } n \geq 2 \end{cases}$$

En C, le type `unsigned int` est généralement codé sur 4 octets, autrement dit 32 bits. Le plus grand entier ainsi représentable est :

$$\texttt{UINT\_MAX} = 2^{32} - 1 = 4294967295$$

Par ailleurs, $f_{47} = 2971215073$. Par conséquent, $\texttt{UINT\_MAX} > f_{47}$. Votre programme devrait être capable de calculer f_{47}.

1. Indiquez le temps nécessaire à votre programme pour calculer puis afficher `fibonacci_aux(47)`.
2. Nous considérons les deux fonctions suivantes

   `fibonacci(n: entier naturel) : entier naturel`
   `fibonacci_aux(n, a , b: entier naturel) : entier naturel`

   qui satisfont les axiomes suivants :

$$\begin{aligned} \texttt{fibonacci\_aux(0,a,b)} &= \texttt{a} \\ \texttt{fibonacci\_aux(1,a,b)} &= \texttt{b} \\ \texttt{fibonacci\_aux(n,a,b)} &= \texttt{fibonacci\_aux(n-1,b,a+b)} \\ \texttt{fibonacci(n)} &= \texttt{fibonacci\_aux(n,0,1)} \end{aligned}$$

   (a) Traduire les axiomes ci-dessus en un programme en C (nom du fichier à déposer sur Arche : `exo3.c`).

   (b) Combien de temps est nécessaire à votre programme pour calculer f_{47}.

Exercice 4 — Les nombres parfaits
Un entier naturel est **parfait** s'il est égal à la somme de ses diviseurs propres. Un diviseur d d'un entier n est dit **propre** si $d < n$.

Le plus petit entier naturel parfait est 6. En effet 1, 2 et 3 sont les diviseurs propres de 6 et 1+2+3=6.

On considère les deux fonctions

- `somme_diviseurs_propres(n: entier naturel) : entier naturel`
- `somme_diviseurs_propres_aux(d, n: entier naturel) : entier naturel`

qui vérifient les axiomes suivants :

[1] `somme_diviseurs_propres_aux(n, n) = 0`

[2] `somme_diviseurs_propres_aux(d, n) = somme_diviseurs_propres_aux(d+1, n)` si d ne divise pas n

[3] `somme_diviseurs_propres_aux(d, n) = d+somme_diviseurs_propres_aux(d+1, n)` si d divise n et $d < n$.

[4] `somme_diviseurs_propres(n) = somme_diviseurs_propres_aux(1, n)`

Traduire les axiomes ci-dessus en un programme en C (nom du fichier à déposer sur Arche : `exo4.c`).