Devoir en temps limité n°2 – 2h

Calculatrices autorisées

On veillera à présenter très clairement sa copie : il faut rédiger les réponses et encadrer les résultats. Pour le code, il doit être indenté, on ne commence pas une fonction en bas de page et on utilise de la couleur pour les commentaires.

Le code doit être commenté dès qu'il dépasse les 5 lignes.

Il est possible (et recommandé) d'utiliser des fonctions auxiliaires en Ocaml. Dans ce devoir **il est interdit d'utiliser les aspects impératifs de Ocaml y compris : le mot-clé mutable, les références, les tableaux et les boucles**

1 Questions de cours

1. Rappeler la définition de $u_n = \Theta(v_n)(n \to +\infty)$ pour $(u_n)_{n\in\mathbb{N}}$ et $(v_n)_{n\in\mathbb{N}}$ deux suites.
2. Voici le nombre d'opérations élémentaires réalisés par 4 fonctions. Déterminer leur complexité asymptotique et classez les fonctions par ordre croissant de rapidité.
   - $f_1 : 4n + 1$
   - $f_2 : 3n^3 + 2n^2$
   - $f_3 : 2n\lfloor \log_{10}(n) \rfloor + n$
   - $f_4 : 3^n \lfloor \log_2(n) \rfloor$
3. Dans le fichier de code suivant, determiner quelles variables sont stockées dans le segment données de la mémoire, dans le segment pile et dans le segment tas. (À priori il y a 7 variables)

```c
#include <stdlib.h>
#include <stdio.h>

int x=0;

int f(int n){
  int* tab = malloc(n*sizeof(int));
  tab[0] = 1;
  tab[1] = 1;
  for(int i=2;i<n;i+=1){
    tab[i] = tab[i-1] +tab[i-2];
  }
  int res = tab[n-1];
  free(tab)
  return res;
}

int main(){
  int t[4] = {1,5,10,14};

  for(int j=0; j<4;j+=1){
    f(t[j]);
  }
}
```

4. Qu'est ce qu'un invariant de boucle ? À quoi ça sert ?
5. Traduire en binaire sur 32 bits le nombre flottant $37,75$. La représentation obtenue est censée être exacte.

2 Ocaml

6. Déterminer le type des fonctions Ocaml suivantes :

```
let f x y = x + y;;
let g a b = if a=3 then b else 1.5;;
let h (x,y,z) = if x=y then z+1 else z;;
let i a b c = let (d,e) = a in
              if b then c
              else d;;
```

7. Définir en Ocaml la fonction `f:float->float` définie par $f(x) = e^x + x^2 - \dfrac{3}{2}$.

8. Définir une fonction Ocaml récursive qui calcule $\sum_{i=0}^{n} 3i^6$ pour n donné en entrée. La signature pourra être `somme6 : int -> int` OU `somme6 : int -> int -> int`.
9. Écrire une fonction récursive `sum : int list -> int` qui calcule la somme des éléments d'une liste d'entiers.
10. Écrire une fonction `nboccurences : 'a list -> 'a -> int` qui prend en entrée une liste l et un élément x et renvoie le nombre de fois où x apparait dans l.
    Par exemple `nboccurences [1;2;3;1;5;1;7;1] 1;;` renvoie 4.
11. Écrire une fonction `del_list : 'a list -> int -> 'a list` qui prend en entrée une liste l et un entier n et renvoie une liste l' qui est l où on a retiré l'élément d'indice n.
    Par exemple `del_list [1;2;3;4] 1;;` renvoie `[1;3;4]` car on supprime le 2, situé à l'indice 1. Dans cette question on considère que les indices commencent à 0.

3 Preuve de programme

On étudie la fonction C suivante :

```c
int mystere(int n){
   assert(n>0);
   int x = 0;
   int y = n;
   while (y!=0){
     x += 3;
     y -= 1;
   }
   return x;
}
```

12. Que renvoie cette fonction ? Écrire sa spécification.
13. Montrer la terminaison de la fonction.
14. Trouver un invariant et montrer la correction de la fonction.

4 Calcul de complexité

On considère les 4 programmes suivants :

```c
void f1(int* tab, int n){
  int m = 3*4;
  for(int i=0; i<n;i+=1){
    tab[i] *= m;
  }
}
```

```c
int f2(int n){
  int res = 1;
  for(int i=0; i<n; i+=1){
    for(int j=0; j<i ; j+=1){
      res *= j;
    }
  }
  return res;
}
```

```c
int f3(int n){
  int res = 1;
  for(int i=0; i<n; i+=1){
    res *= i;
    for(int j=0; j<i ; j+=1){
      res *= j;
    }
  }
  return res;
}
```

```c
int f4(int n){
  int res = 1;
  for(int i=0; i<n; i+=1){
    for(int j=i+3; j<=i+6 ; j+=1){
      res *= 3;
    }
  }
  return res;
}
```

15. Compter **exactement** combien de multiplications sont réalisées par chaque fonction.
16. En déduire la complexité asymptotique des quatres fonctions.

On considère la fonction Ocaml suivante qui calcule 2^n de manière un peu bête :

```
let rec puissance_2_bete n =
   if n = 0 then 1
   else if n mod 2 = 0 then (puissance_2_bete (n/2)) * (puissance_2_bete (n/2))
   else 2 * (puissance_2_bete (n/2)) * (puissance_2_bete (n/2));;
```

On note $C(n)$ la complexité de la fonction sur l'entrée n. On rappelle qu'en Ocaml, le calcul `n/2` donne en réalité $\lfloor \frac{n}{2} \rfloor$.

17. Déterminer la formule de récurrence de la suite $(C(n))_{n\in\mathbb{N}}$.
18. Pour $n = 2^k$, trouver la forme générale de la suite. Il n'est pas nécessaire de généraliser à n qui n'est pas une puissance de 2.
19. Montrer la terminaison de cette fonction.
20. Montrer la correction de cette fonction. On remarquera qu'elle est récursive.

5 Suite de Syracuse

La suite de Syracuse est définie par récurrence par un $u_0 \in \mathbb{N}$ et

$$\forall n \in \mathbb{N}, \quad u_{n+1} = \begin{cases} u_n/2 & \text{si } u_n \text{ est pair} \\ 3 * u_n + 1 & \text{si } u_n \text{ est impair} \end{cases}$$

Par exemple, avec $u_0 = 13$, on obtient la suite 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, 4, 2, 1, 4, 2, . . .

Après avoir atteint le nombre 1, la suite de valeurs 1, 4, 2 se répète indéfiniment en un cycle de longueur 3 appelé cycle trivial. La conjecture de Collatz est l'hypothèse selon laquelle la suite de Syracuse de n'importe quel entier u_0 strictement positif atteint toujours 1. Bien qu'elle ait été vérifiée pour les 5,7 premiers milliards de milliards d'entiers et en dépit de la simplicité de son énoncé, cette conjecture défie depuis de nombreuses années les mathématiciens.

Dans cette exercice, on supposera que la conjecture est vraie et que les fonctions qu'on écrit termineront toujours.

21. Écrire une fonction `suivant : int -> int` telle que `suivant u_n` calcule le terme u_{n+1} en fonction du terme u_n = `u_n`.
22. En déduire une fonction `syracuse : int -> int -> int` qui prend en entrée u_0 et n et renvoie le terme u_n de la suite de Syracuse de premier terme u_0.

On appelle orbite de u_0 la liste des termes de la suite de Syracuse de u_0 jusqu'à ce que l'on tombe sur 1 (inclus). L'orbite de 13 est donc 13, 40, 20, 10, 5, 16, 8, 4, 2, 1.

Plusieurs caractéristiques d'une orbite peuvent être intéressantes :

- son temps de vol correspond au nombre total d'entiers visités sur l'orbite ;
- son altitude est donnée par le plus grand entier visité sur l'orbite ;
- son temps de vol en altitude correspond au nombre d'étapes avant de passer strictement en-dessous du nombre de départ ;
- son temps de vol avant la chute correspond au nombre d'étapes minimum après lequel on ne repasse plus au-dessus de la valeur de départ.

Par exemple pour l'orbite du nombre 13, l'altitude vaut 40 (maximum de la suite), le temps de vol vaut 10, le temps de vol en altitude vaut 3 (on passe à $10 < 13$ lors de la 3^e étape) et le temps de vol avant la chute vaut 6 (on passe à $8 < 13$ lors de la 6^e itération et ensuit on ne repasse jamais au-dessus).

23. Écrire une fonction `temps_de_vol : int -> int` qui prend en entrée u_0 et renvoie le temps de vol correspondant à l'orbite de u_0.
24. Écrire une fonction `altitude : int -> int` qui prend en entrée u_0 et renvoie l'altitude de l'orbite de u_0.
25. Écrire une fonction `temps_en_altitude : int -> int` qui prend en entrée u_0 et renvoie le temps de vol en altitude correspondant à l'orbite de u_0 avec $u_0 > 1$.
26. Écrire une fonction `temps_avant_chute u_0` qui prend en entrée u_0 et renvoie le temps de vol avant la chute pour l'orbite de u_0 avec $u_0 > 1$.