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.
   - $f_1 : 4n + 1 = \Theta(n)$
   - $f_2 : 3n^3 + 2n^2 = \Theta(n^3)$
   - $f_3 : 2n\lfloor\log_{10}(n)\rfloor + n = \Theta(n\log(n))$
   - $f_4 : 3^n\lfloor\log_2(n)\rfloor = \Theta(3^n\log(n))$

   La complexité de f_1 est la meilleure, suivie de f_3, f_2 et f_4.
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]);
  }
}
```

   - x est stocké dans le segment donnée, c'est une variable globale
   - tab est alloué dans le tas (plus exactement $*tab$ est dans le tas et la variable tab est dans la pile, mais il est acceptable de confondre la mémoire allouée et le pointeur)
   - t est un tableau statique et est donc alloué dans la pile.
   - Toutes les autres variables sont locales et donc allouées dans la pile.
4. Qu'est ce qu'un invariant de boucle ? À quoi ça sert ?
   Un invariant de boucle est une propriété ou un affirmation qui est vraie avant la boucle et reste vraie à chaque itération de la boucle. Un invariant bien choisi peut permettre de montrer la correction d'un code.
5. Traduire en binaire sur 32 bits le nombre flottant 37, 75. La représentation obtenue est censée être exacte.
   On applique l'algorithme habituel : 0 | 10000100 | 00101110000000000000000

2 Ocaml

6. Déterminer le type des fonctions Ocaml suivantes :

```
let f x y = x + y;; (*int -> int -> int*)
let g a b = if a=3 then b else 1.5;; (*int -> float -> float*)
let h (x,y,z) = if x=y then z+1 else z;; (*'a*'a*int -> int*)
let i a b c = let (d,e) = a in (*'a*'b -> bool -> 'a -> 'a*)
              if b then c
              else d;;
```

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

```
f x = exp(x)+.x**2.-.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`.

```
let rec somme6 n = match n with
  |0 -> 0
  |_ -> 3*n*n*n*n*n*n + somme6 (n-1);;
```

9. Écrire une fonction récursive `sum : int list -> int` qui calcule la somme des éléments d'une liste d'entiers.

```
let rec sum l = match l with
  |[] -> 0
  |t::q -> t + sum q;;
```

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.

```
let nboccurences l x = match l with
  |[] -> 0
  |t::q when t=x -> 1 + nboccurences q x
  |t::q -> nboccurences q x;;
```

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.

```
let rec del_list l i = match l,i with
  |[],_ -> failwith "pas assez d'éléments
  |t::q,0 -> q
  |t::q, _ -> del_list q (i-1);;
```

3 Preuve de programme

On étudie la fonction C suivante :

```
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.
On remarque que la boucle while s'effectue n fois et qu'à chaque tour x augmente de 3. x vaut donc $3 * n$ à la fin.
La spécification est : Précondition : $n > 0$ Postcondition : Renvoie $3 * n$

13. Montrer la terminaison de la fonction.
    y est un invariant de la fonction. En effet :
    - C'est un entier
    - Il diminue strictement (de 1) à chaque tour de la boucle
    - Il est positif au début lorsque n est une entrée valide et reste positif car sinon la boucle s'arrête.

14. Trouver un invariant et montrer la correction de la fonction.
    Un invariant serait : "à chaque itération de la boucle, x vaut $3 * (n - y)$".
    Cet propriété est vraie avant la boucle, $x = 0 = 3 * (n - n)$.
    Supposons la propriété vraie à la fin d'une des itérations. Alors au début de l'itération suivante $x = 3 * (n - y)$.
    La nouvelle valeur de x est $x' = x + 3$. La nouvelle valeur de y est $y' = y - 1$.
    On a donc $3 * (n - y') = 3 * (n - y) + 3 = x + 3 = x'$. Notre propriété est un invariant.
    À la fin du programme, lorsque $y = 0$, l'invariatn nous permet d'écrire que $x = 3 * (n - 0) = 3n$. CQFD.

4 Calcul de complexité

On considère les 4 programmes suivants :

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

```
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;
}
```

```
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;
}
```

```
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.
    - Pour $f1$: $1 + \sum_{i=0}^{n-1} 1 = n + 1$.
    - Pour $f2$: $\sum_{i=0}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$
    - Pour $f3$: $\sum_{i=0}^{n-1} (1 + \sum_{j=0}^{i-1} 1) = \sum_{i=0}^{n-1} (1 + i) = n + \frac{n(n-1)}{2}$
    - Pour $f4$: $\sum_{i=0}^{n-1} \sum_{j=i+3}^{i+6} 1 = \sum_{i=0}^{n-1} 4 = 4n$

16. En déduire la complexité asymptotique des quatres fonctions.
    $f1$ et $f4$ sont linéaires, $f2$ et $f3$ sont quadratiques.

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 $(C(n))_{n\in\mathbb{N}}$.
$C(0) = 1$ et $\forall n \in \mathbb{N}^*, C(n) = 2C(n/2) + 2$
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.

On devine que pour $l \leq k$, $C(n) = 2^l C(n/2^l) + \sum_{i=1}^{l} 2^i$.

Montrons le par récurrence : Pour $n = 0$, $C(n) = 1 * C(n) + 0$

Supposons que ce soit vrai pour $l < k$: $C(n) = 2^l C(n/2^l) + \sum_{i=1}^{l} 2^i$.

En utilisant la formule de récurrence de C sur $C(n/2^l)$ il vient :

$$\begin{aligned} C(n) &= 2^l C(n/2^l) + \sum_{i=1}^{l} 2^i \\ &= 2^l (2C(n/2^{l+1}) + 2) + \sum_{i=1}^{l} 2^i \\ &= 2^{l+1} C(n/2^{l+1}) + 2^{l+1} + \sum_{i=1}^{l} 2^i \\ &= 2^{l+1} C(n/2^{l+1}) + \sum_{i=1}^{l+1} 2^i \end{aligned}$$

En particulier c'est vrai en $l = k$ et :

$$\begin{aligned} C(n) &= 2^k C(n/2^k) + \sum_{i=1}^{k} 2^i \\ &= n * C(1) + \sum_{i=1}^{k} 2^i \\ &= n * (2C(0) + 2) + \frac{1 - 2^{k+1}}{1 - 2} \\ &= 4n + 2^{k+1} - 1 \\ &= 6n - 1 \end{aligned}$$

La complexité semble donc être linéaire.

19. Montrer la terminaison de cette fonction.
Un variant des appels récursifs est n. En effet :
   (a) C'est un entier
   (b) Un n négatif n'est pas une entrée admissible de la fonction.
   (c) Si $n \neq 0$, alors le quotient de la division euclidienne de n par 2 est bien strictement inférieur à n.
20. Montrer la correction de cette fonction. On remarquera qu'elle est récursive.
Comme elle est récursive, il suffit de montrer que son cas de base est correct et qu'en supposant que les appels récursifs effectués sont corrects, les autres cas sont corrects.
Si $n = 0$ on renvoie $1 = 2^0$.
Si $n = 2k$ est pair, en supposant que `puissance_2_bete (n/2)` renvoie $2^{n/2} = 2^k$, alors on renvoie $2^k * 2^k = 2^{2k} = 2^n$.
Si $n = 2k+1$ est pair, en supposant que `puissance_2_bete (n/2)` renvoie $2^{n/2} = 2^k$, alors on renvoie $2 * 2^k * 2^k = 2^{2k+1} = 2^n$.

5 Suite de Syracuse

21.

```
let suivant u_n =
  if u_n mod 2 = 0 then
    u_n / 2
  else
    3 * u_n + 1
```

22. On calcule récursivement u_{n-1} et on cherche le suivant :

```
let rec syracuse u_0 n =
if n = 0 then
u_0
else
suivant (syracuse u_0 (n - 1))
```

On peut également caculer u_1 puis appliquer récursivement la fonction en prenant ce u_1 comme nouveau u_0 :

```
let rec syracuse u_0 n =
if n = 0 then
u_0
else
syracuse (suivant u_0) (n - 1)
```

23. Pour calculer le temps de vol à partir de u_0, on peut calculer le temps de vol de u_1 et ajouter une étape :

```
let rec temps_de_vol u_0 =
  if u_0 = 1 then
    1
  else
    1 + temps_de_vol (suivant u_0)
```

24. Pour calculer l'altitude maximale à partir de u_0, on peut calculer l'altitude maximale à partir de u_1 puis prendre le maximum entre ce que l'on obtient et u_0 :

```
let rec altitude u_0 =
  if u_0 = 1 then
    1
  else
    max u_0 (altitude (suivant u_0))
```

25. On va écrire une fonction auxiliaire locale `vol : int -> int` qui va être similaire à `temps_de_vol` mais qui va s'arrêter dès que l'on tombe en dessous de l'altitude initiale plutôt que lorsque l'on arrive sur 1.

```
let temps_en_altitude u_0 =
    let rec vol u =
      if u < u_0 then
        0
      else
        1 + vol (suivant u)
    in
      vol u_0
```

26. Comme pour la question précédente, comme il est nécessaire de mémoriser le tout permier u_0, on va écrire une fonction auxiliaire locale qui calcule le temps de chute à partir d'un terme arbitraire u. Si on arrive sur 1 c'est terminé, il n'y a plus d'étapes et donc le temps avant chute est 0.
Sinon, on calcule le temps avant chute du terme suivant. S'il est strictement positif, c'est que l'on repassera en dessous de la valeur de départ et on a trouvé le temps de chute en ajoutant une étape. Sinon, c'est que l'on ne repassera plus en dessous de u_0. Si $u \geqslant u_0$ alors il y a une étape pour passer en dessous de u_0 et sinon le temps de chute pour ce terme u est nul.

```
let temps_avant_chute u_0 =
  let rec trouve_tps u =
    if u = 1 then
      0
    else
      let tps_chute = trouve_tps (suivant u) in
        if tps_chute > 0 then
          tps_chute + 1
        else if u >= u_0 then
            1
        else
            0
      in
        trouve_tps u_0
```

Une autre possibilité est d'écrire une fonction auxiliaire qui mémorise le nombre d'étapes effectuées et la valeur estimée du temps de chute et qui les met à jour.

Comme on ne peut pas modifier des valeurs, on va passer ces deux valeurs en paramètres et calculer le couple formé par ces valeurs mises à jour.

L'appel `vol u nb_etapes tps_chute` continue de calculer à partir d'un terme `u` le nombre d'étapes nécessaires pour atteindre 1 et le temps de vol avant chute, sachant que l'on a déjà effectué `nb_etapes` et que le temps de chute après vol vaut `tps_chute` pour l'instant.

```
let temps_avant_chute u_0 =
  let rec vol u nb_etapes tps_chute =
    if u = 1 then
      (nb_etapes, tps_chute)
    else if u >= u_0 then
      vol (suivant u) (nb_etapes + 1) (nb_etapes + 1)
    else
      vol (suivant u) (nb_etapes + 1) tps_chute
  in
    snd (vol u_0 0 0)
```