Table des matières

Premier devoir surveillé d'informatique

1 Exercice du TD

Exercice 1. Fibonacci efficace

Soit $F \in \mathbb{N}^{\mathbb{N}}$ telle que $F_0 = 0, F_1 = 1$ et $\forall n \in \mathbb{N}$, $F_{n+2} = F_{n+1} + F_n$.

On a vu en cours que la programmation récursive naïve de cette suite est inefficace. On propose ci-dessous deux méthodes pour résoudre ce problème. Pour chacune on aura besoin de définir une fonction auxiliaire.

1. **Fonction auxiliaire qui utilise un argument supplémentaire :**
   On utilise une fonction `fiboAux` qui prend en entrée deux termes consécutifs f_k et f_{k+1} et un entier n, et qui renvoie f_{k+n}.
2. **Fonction auxiliaire qui renvoie une valeur supplémentaire :**
   On utilise une fonction auxiliaire `fiboAux` qui renvoie non pas seulement F_n, mais le couple (F_n, F_{n+1}).

1. Écrire les deux fonctions correspondantes.
2. Démontrer la correction et calculer la complexité de la deuxième méthode.

solution :

2 Méthode chinoise pour le calcul d'un pgcd (d'après E3A 2018)

Tout entier naturel non nul n s'écrit de manière unique $n = 2^v k$ avec v un entier naturel (qui peut valoir zéro) et k un entier naturel impair. Dans la suite de cet exercice, nous appelons *valuation* de n l'entier v et *résidu* de n l'entier k. Par convention, le résidu de zéro est zéro et sa valuation aussi.

1. Expliquer succinctement comment, à partir de l'écriture en base deux de $n \in N^*$, on peut lire la valuation et le résidu de n.
2. L'entier 192 s'écrit en base deux 11000000. Donner sa valuation et son résidu.
   solution : La valuation de 192 est 6, et son résidu s'écrit 11 en base deux, donc 3 en base 10.
3. Écrire en Caml une fonction `residu : int -> int` qui prend en argument un entier positif ou nul et renvoie son résidu.
   solution :

```
let rec residu n =
  if n =0 then 0
  else if n mod 2 = 0 then residu (n/2)
  else n
;;
```

4. Écrire une fonction `residu_et_valuation` de type `int -> (int * int)` qui prend un entier et renvoie (son résidu, sa valuation). Pour soucis d'efficacité, ne pas utiliser la fonction `residu`.
   solution :

```
let rec residu_et_valuation n =
  if n=0 then 0,0
  else  if n mod 2 = 0 then
    let r,v = residu_et_valuation (n/2) in
    r, v+1
  else
```

```
    n, 0
;;
```

5. Démontrer que pour tout $(a,b) \in (\mathbb{N}^*)^2$, $a \wedge b = \min(a,b) \wedge (|a-b|)$. On pourra se contenter de traiter le cas où $a > b$.
   solution : Dans le cas précisé par l'énoncé, il s'agit de prouver que $a \wedge b = b \wedge (a-b)$. Notons $d_1 = a \wedge b$ et $d_2 = b \wedge (a-b)$.
   - L'entier d_1 divise a et b donc $a-b$. C'est donc un diviseur commun à b et $a-b$, donc il divise d_2.
   - De même, d_2 divise b et $a-b$, donc $b+a-b$ c'est-à-dire a. Donc c'est un diviseur commun à a et b, donc il divise d_1.

   Enfin, d_1 et d_2 étant de même signe, on déduit $d_1 = d_2$.
   Remarque : En fait il s'agit du lemmen d'Euclide, « pour $q = 1$ ».

Pour calculer le pgcd (plus grand commun diviseur) de deux entiers, nous pouvons utiliser l'algorithme des soustractions successives décrit ci-après.

Entrée : deux entiers naturels non nuls a et b
tant que *b est non nul* **:**
  Remplacer b par |a − b|.
  Remplacer a par le minimum de a et de l'ancienne valeur de b.
fin
Renvoyer a

On rappelle qu'en OCaml, le minimum de deux nombres s'obtient par la fonction `min`, et la valeur absolue d'un entier par la fonction `abs`.

6. Écrire en Caml une fonction récursive `pgcd2 : int -> int -> int` qui calcule le pgcd de deux entiers naturels non nuls en utilisant la méthode des soustractions successives.
   solution :

```
let rec pgcd2 a b =
  if b=0 then a
  else
    pgcd2 (min a b) (abs (a-b))
;;
```

7. Estimer (en justifiant) la complexité de cet algorithme en fonction de $n = max(a,b)$.
   solution : À chaque appel récursif, $\min(a,b)$ diminue au mois de 1. D'où une complexité en $O(n)$ (chaque appel coûte $O(1)$).
   Par ailleurs le maximum de n appels récursifs est atteint lorsque a ou b vaut 1.

La méthode chinoise, mentionnée dans Les Neuf Chapitres sur l'art mathématique écrit aux débuts de la dynastie Han, consiste à remplacer la ligne 2 par la ligne suivante :

Remplacer b par le résidu de $|a-b|$.

On admet que cette méthode permet de calculer le pgcd de a et b si a ou b est impair. En particulier ceci pourra être utiliser pour répondre aux questions suivantes.

8. Écrire une fonction récursive `pgcd_chinois : int -> int -> int` qui calcule le pgcd de deux entiers dont l'un au moins est impair grâce à la méthode chinoise.
   solution :

```
let rec pgcdChinois a b =
  if b=0 then a
  else
    pgcdChinois (min a b) (residu (abs (a-b)))
;;
```

9. Estimer (en justifiant) la complexité de cette méthode en fonction de $n = max(a,b)$, dans le cas où a et b sont tous les deux impairs.
   solution : Supposons a et b impairs et distincts. Alors $\min(a,b)$ est impair, de même que le résidu de $|a-b|$ puisqu'un résidu est toujours impair. On prouve ainsi par récurrence qu'à chaque appel récursif les arguments de `pgcdChinois` serot impairs.

De plus si a et b sont impairs alors $|a-b|$ est pair donc son résidu est au plus $\frac{|a-b|}{2}$.

Ainsi à chaque appel récursif la quantité $n = \max(a,b)$ diminue au moins de moitié. Donc le nombre d'étapes sera en $O(\log_2 n)$.

10. Calcul de puissance :
    (a) Soit $x \in \mathbb{Z}$ et $n \in \mathbb{N}$. Soit $q = \left\lfloor \frac{n}{2} \right\rfloor$ et $y = x^q$. Exprimer x^n en fonction de y. On pourra distinguer les cas où n est pair ou impair.
    (b) En déduire une fonction récursive `puissance` prenant deux entiers x et n et renvoyant x^n.
    (c) (bonus) L'opérateur `lsl` de OCaml prend deux entier x et i et renvoie l'entier obtenu en décalant les chiffres de x de i cases vers la gauche (« lsl » signifie « logical shift left »). Il s'agit d'un opérateur infixe, ce qui signifie qu'il se place entre ses arguments. Par exemple `1 lsl 2` renvoie 4.
    En utilisant `lsl`, écrire en une ligne une fonction pour calculer des puissances de 2.
    solution :

```
let puissance2 n=
  (* Renvoie 2^n *)
  1 lsl n;;
```

11. Traitons le cas général. Pour tout entier $k \in \mathbb{N}$, sa valuation sera notée $\sigma(k)$. Soit $(a,b) \in (\mathbb{N}^*)^2$. Soit $v = \min\big(\sigma(a), \sigma(b)\big)$.
    (a) Démontrer que $a \wedge b = 2^v \left(\frac{a}{2^v} \wedge \frac{b}{2^v}\right)$.
    solution : Par définition de v, $\frac{a}{2^v}$ et $\frac{b}{2^v}$ sont entiers. Dès lors la formule $\forall (a,b,c) \in \mathbb{Z}^3$, $a \times (b \wedge c) = ab \wedge ac$ qui figure dans le cours de maths permet de conclure.
    (b) Démontrer que le pgcd de $\frac{a}{2^v}$ et $\frac{b}{2^v}$ peut être calculé par `pgcd_chinois`.
    solution : Par définition de v, l'un des deux nombres $\frac{a}{2^v}$ ou $\frac{b}{2^v}$ est impair, donc `pgcd_chinois` est utilisable.
    (c) En déduire une fonction `pgcd_final` permettant de calculer à l'aide de la méthode chinoise le pgcd de n'importe quels entiers a et b.
    solution :

```
let pgcd_final a b =
  let ra,va = residu_et_valuation a
  and  rb,vb = residu_et_valuation b in
  let v = min va vb in
  puissance2 v * pgcdChinois (min a b ) (residu (abs(a-b)))
;;
```

3 Puissances entières (E3A 2015)

Cet exercice utilise la fonction `puissance` de l'exercice précédent. On rappelle que la complexité de celle-ci a été vue en DM.

On dit qu'un entier n est une *puissance entière* lorsqu'il existe $(m,k) \in [\![2, \infty[\![^2$ tels que $n = m^k$.

1. Écrire une fonction `test_puissance : int -> int -> bool` prenant deux entiers n et k tous deux > 1 et indiquant s'il existe m tel que $n = m^k$.
   solution :

```
let test_puissance n k =

  let rec aux m =
    let essai = puissance m k in
    if essai > n then false
    else if essai = n then true
    else aux (m+1)
  in

  aux 2
;;
```

2. (a) Soit $n \in \mathbb{N}$. On suppose que n est une puissance entière, soit $(m,k) \in [\![2,\infty[\![^2$ tel que $n = m^k$. Justifier que $k \leqslant \log_2(n)$.
solution :

```
let test_puissance_entiere n =
  let rec boucle k = (* Je fais cette boucle à l'envers. De la sorte je m'arrête
    ↪ lorsque k=1, ça m'évite le calcul du log à chaque étape.*)
    if k=1 then false
    else test_puissance n k || boucle (k-1)
  in
  boucle (int_of_float (log (float_of_int n) /. log 2. ))
;;
```

(b) Écrire une fonction `test_puissance_entiere : int -> bool` qui prend un entier $n > 1$ et qui indique si n est une puissance entière.
solution :

```
let test_puissance_entiere n =
  let rec boucle k = (* Je fais cette boucle à l'envers. De la sorte je m'arrête
    ↪ lorsque k=1, ça m'évite le calcul du log à chaque étape.*)
    if k=1 then false
    else test_puissance n k || boucle (k-1)
  in
  boucle (int_of_float (log (float_of_int n) /. log 2. ))
;;
```

(c) Rappeler sans démonstration la complexité de la fonction `puissance`. Démontrer que la complexité de `test_puissance_entiere n` est en $O\left(n\log_2(n)\log_2(\log n)\right)$.
solution :

- `test_puissance n k` a une complexité en $O(n\log k)$, or k sera toujours $\leqslant \log_2(n)$. Donc la complexité de `test_puissance` est en $O\left(n\log(logn)\right)$.
- L'appel `test_puissance_entiere n` lance `puissance_entiere n k` pour tout $k \in [\![2, \lfloor \log_2(n) \rfloor [\![$, donc la complexité totale est en $O\left(\log(n) \times n\log(logn)\right)$.

(d) En déduire une fonction `liste_puissance_entiere : int -> int list` prenant un entier n et renvoyant la liste des entiers de $[\![2,n[\![$ qui sont des puissances entières.
solution :

```
(* Pas besoin de fonction aux ici.*)
let rec liste_puissance_entiere n =
  if n <= 3 then []
  else
    (if test_puissance_entiere n then [n] else [])
    @ liste_puissance_entiere (n-1)
;;
```

3. En vous inspirant du crible d'Érathosthène écrire une autre version plus efficace de la fonction `liste_puissance_entiere` ↪ . Comme l'utilisation de tableaux en Caml n'a pas encore étudiée en cours, vous pouvez rédiger cette fonction en Python.
solution : L'idée est de créer un tableau `puissanceEntiere` de n cases rempli de booléens. À terme, pour tout $i \in [\![2,n[\![$, `puissanceEntiere[i]` vaudra `True` ssi i est une puissance entière.
On parcourt tous les m et tous les k et on marque `puissanceEntiere[m**i]` comme `True`.

```
let rec intervalle a b =
  if a>= b then [] else a::intervalle (a+1) b
;;
let liste_puissance_opti n =
  let est_puissance_entiere = Array.make (n+1) false in (* À la fin, est_puissance_entiere
    ↪ .(i) vaudra true ssi i est une puissance entière *)

  for k = 2 to (int_of_float (log (float_of_int n) /. log 2. )) do
    let fini = ref false
    and m = ref 2 in
    while not !fini do
```

```
      let essai = puissance !m k in
      (if essai < n then est_puissance_entiere.(essai) <- true
       else fini := true);
      incr m
    done
  done;

  List.filter
    (fun i -> est_puissance_entiere.(i))
    (intervalle 2 (n+1))
;;
```

4 Un peu de listes

1. Écrire une fonction `egales` prenant des listes l_1 et l_2 et indiquant si $l_1 = l_2$.
2. Écrire une fonction `a_lenvers` prenant une liste l et renvoyant la liste contenant les mêmes éléments mais dans l'ordre inverse.
3. (***) Écrire une fonction `sommePossible` prenant une liste d'entiers strictement positifs l et un entier n et indiquant s'il est possible d'obtenir n en sommant des éléments de l. Les répétitions sont autorisées. *Indication :* Si `l=t::q`, il y a deux possibilités pour obtenir n en sommant des éléments de l :
   - ne pas utiliser t ;
   - ou utiliser t.