Option Informatique
Complexité

Corrigé

21 octobre 2007

1 Décodage d'entiers

Question 1 *Écrire une fonction* `valeur s` *qui renvoie l'entier associé à s.*

```
let rec valeur b s =
  let n = string_length s in
  if n = 1 then
    int_of_string s
  else
    let head = sub_string s (n-1) 1 and
        tail = sub_string s 0 (n - 1) in
    int_of_string head + b * (valeur b tail) ; ;
```

Question 2 *Écrire une fonction* `compare s s'` *qui renvoie la valeur* `true` *quand $s \geq s'$. On ne calculera pas les entiers associés à* `s` *et* `s'` *car cette démarche est caduque si les entiers manipulés sont trop grands pour le type* `int`*.*

Notons ici qu'il faut bien faire attention au cas d'arrêt lorsque les deux nombres comparés sont égaux !

```
let rec compare s s' =
  let n = string_length s and
      n' = string_length s' in
  if n <> n' then
    n > n'
  else
    let h = sub_string s 0 1 and
        h' = sub_string s' 0 1 in
    if h <> h' then
      (int_of_string h > int_of_string h')
    else
      if n = 1 then true else
        compare (sub_string s 1 (n-1))
               (sub_string s' 1 (n-1)) ; ;
```

Question 3 *Évaluer la complexité de ces fonctions.*

La fonction `valeur` termine, par récurrence sur la longueur de son paramètre s au fil des appels récursifs. Le même argument montre qu'elle fait un nombre *linéaire* ($\mathcal{O}(n)$, où $n = |s|$) d'appels.

Pour la fonction `compare`, on raisonne sur le *pire cas* pouvant se produire (chaînes égales), par récurrence sur la longueur `n` des chaînes argument `s` et `s'`, et on montre que la fonction termine en $\mathcal{O}(n)$ appels.

2 Représentation d'un nombre en base b

Question 4 *a*

Donner une fonction `calcul_base_min` *telle que* `calcul_base_min n b` *affiche la représentation en base b de n.*

b Vérifier votre précédure à l'aide de quelques exemples en base 10.

c Soit n un entier dont on connaît la réprésentation en base b. Comment fait-on pour connaître facilement le quotient et le reste dans la division euclidienne de n par b ?

```
let rec calcul_base_min b n=
  if n < b then string_of_int n else
    let r = (n mod b) in
      calcul_base_min b ((n-r)/b) ^ string_of_int r ; ;

#calcul_base_min 10 42 ; ;
- : string = "42"
#calcul_base_min 10 100000 ; ;
- : string = "100000"
#calcul_base_min 10 9 ; ;
- : string = "9"

let rec myrem b n = if n < b then n else myrem b (n - b) ; ;

let myquo b n = (n - myrem b n) / b ; ;
```

Question 5 *Évaluer la complexité de la fonction précédente.*

On considère le logarithme en base b du contenu de la variable `binary` : on voit aisément qu'il diminue de 1 à chaque passage dans la boucle. La complexité de cette fonction est donc logarithmique ($\mathcal{O}(\ln_b(n))$).

3 Application : la multiplication rapide par un entier

Question 6 *On pose $u_n = nx$. on a donc $u_0 = 0$ et pour tout $n \geq 0$, $u_{n+1} = u_n + x$.*

a Écrire une fonction `produit` *telle que* `produit n x` *renvoie nx, et qui ne fait que des additions.*

b Quelle est sa complexité ?

```
let rec produit n x =
  if n = 0 then 0 else x + produit (n-1) x ; ;
```

La fonction est de complexité linéaire, par récurrence sur `n`.

Question 7 *Montrer que $v_p = nx$.*

L'algorithme consiste à parcourir la représentation binaire de n, des poids faibles aux poids forts, et à ajouter x multiplié par la puissance de 2 correspondant au bit courant si celui-ci vaut 1.

Si $n = [t_{p-1} \dots t_0]_2 = \Sigma_{k=0}^{p-1} 2^k t_k$, alors $nx = \Sigma_{k=0}^{p-1} 2^k t_k x$. Il suffit donc de prouver par récurrence forte sur n que $\forall i,\ 1 \leq i \leq p$, $v_i = \Sigma_{k=0}^{i-1} y_k t_k$ **et** $y_i = 2^i x$, ce qui se fait aisément.

Question 8 *En déduire une fonction* `produit_rapide` *telle que* `produit_rapide n x` *renvoie nx et qui ne fait que des additions. On calculera t_i au fur et à mesure du calcul des y_i et v_i.*

```
let rec produit_rapide n x =
  let v = ref 0 and y = ref x and binary = ref n in
  while (!binary > 1) do
    if (!binary mod 2) = 1 then v :=!v +!y;
    y :=!y +!y;
    binary :=!binary/2;
  done;
  if (!binary mod 2) = 1 then v :=!v +!y;
 !v;;
```

Notez qu'il faut bien faire attention à la dernière addition pour le bit de poids fort !

Question 9 *Evaluer la complexité de cette fonction.*

On considère le logarithme en base 2 du contenu de la variable `binary` : on voit aisément qu'il diminue de 1 à chaque passage dans la boucle. La complexité de cette fonction est donc logarithmique ($\mathcal{O}(\ln_2(n))$).

4 Calcul du PGCD

Question 10 *Donner une fonction* `calcul_fibo` *qui vous permette d'obtenir le cinquantième terme de la suite de Fibonacci en moins de vingt secondes.*

```
(* récursive avec astuce*)
let rec calcul_fibo_r a b n =
    if n = 0 then a
    else if n = 1 then b
    else calcul_fibo_r b (a+b) (n-1);;

(* itérative avec stockage*)
let rec calcul_fibo_i n =
  let u = ref 1 and v = ref 1 and tmp = ref 0 in
  for i = 2 to n+1 do
    tmp :=!u;
    u :=!v;
    v :=!tmp+!u;
  done;
 !u;;
```

Question 11 *Évaluer la complexité de cette fonction.*

Dans sa version récursive, on prouve que la fonction est de complexité linéaire par récurrence descendante sur n.

Dans sa version itérative, on obtient le même résultat en considérant simplement le nombre de passages dans la boucle indicée par `i`.

On introduit l'environnement suivant : Env $= n, p :$ `int`, et la propriété $H =$"$n \geq p \geq 0$ et $n \wedge p = a \wedge b$".

Question 12 *a Quelle valeur suffit-il de donner initialement à n et p pour avoir H vraie ?*

b On suppose $p = 0$. Quel est, dans chaque cas, le PGCD de n et p ?

c On suppose que p est un entier strictement positif.

a Initialement, on fixe n au maximum de a et b, et p au minimum de ces deux nombres.

b Dès lors, si p vaut 0, $n \wedge p = n$, soit a ou b selon que, respectivement, b ou a est le plus petit de ces deux nombres.

c Si p est strictement positif, on applique l'algorithme d'euclide, et on calule le pgcd de p avec le reste de la division euclidienne de n par p. Les valeurs de n et p sont mises à jour respectivement avec ces deux derniers nombres.

Question 13 *En déduire une écriture itérative puis récursive de la fonction* `pgcd(a,b)`*. Justifier de son arrêt et de sa validité.*

```
let pgcd a b =
    let n = ref (max a b) and p = ref (min a b) in
    while (!p >= 0) do
      let t =!p in begin
      p :=!n mod!p;
      n :=!t;
      end;
    done;
   !n;;

let rec pgcd a b =
  if a < b then pgcd b a
  else if b = 0 then a
       else let  r = a mod b in
            pgcd b r;;
```

Question 14 *On étudie dans cette question la version récursive. On introduit la suite de Fibonnaci (f_n)<>.*

a On suppose que $a \geq b > 0$ Établir que si `pgcd(a,b)` *fait n appels récursifs avec $n \geq 1$ alors $a \geq f_n$ et $b \geq f_{n-1}$.*

b Montrer que $f_n \geq (\frac{1+\sqrt{5}}{2})^{n-1}$

c En déduire la complexité de la version récursive est en $O(\ln b)$. Que dire de la complexité de la version itérative ?

a Montrons que si $a \geq b > 0$ et si `gcd_rec(a,b)` fait $n \geq 1$ appels récursifs, alors $a \geq f_n$ et $b \geq f_{k-1}$ par récurrence sur n. On fonde l'induction pour $n = 1$ en disant qu'alors $b \geq 1 = f_0$ et $a \geq 1 = f_1$.

Comme $b > (a \mathsf{mod} b)$, dans chaque appel récursif l'hypothèse selon laquelle $a \geq b$ est conservée.

Supposons que la propriété est vraie si $n-1$ appels sont effectués : puisque $n > 0$ on a $b \geq 1$ et `gcd_rec(a,b)` appelle `gcd_rec(b, a mod b)` récursivement, qui effectue à son tour $n-1$ appels récursifs. On a donc $b \geq f_n$ par hypothèse de récurrence, et $a \mathsf{mod} b \geq f_{n-1}$. On a $b + (a \mathsf{mod} b) = b + (a - \lfloor a/b \rfloor b) \leq a$. On en tire ainsi $a \geq f_{k-1} + f_k$, d'où le résultat.

b Ceci se montre sans difficulté par récurrence sur n. On rappelle la formle de Binet :

$$f_n = \frac{(1+\sqrt{5})^n - (1-\sqrt{5})^n}{2^n \sqrt{5}}$$

c On tire le résultat des deux questions précédentes. La version itérative a la même complexité que la version récursive, si l'on considère le nombre d'affectations à la place du nombre d'appels récursifs comme mesure de complexité.

5 Algorithme d'Euclide étendu

Question 15 *1. Montrer que pout tout $O \leq k \leq n$ nous avons $au_k + bv_k = r_k$*

2. Montrer que pour $k \geq 0$ nous avons $u_{2k} \geq 0$ et $u_{2k+1} \leq 0$

a On raisonne simultanément sur M et N, pour montrer que

$$\begin{cases} M_k[1] = M_k[2]a + M_k[3]b \\ N_k[1] = N_k[2]a + N_k[3]b \end{cases}$$

On a, en sortie d'un tour de boucle :

$$M_k[1] = qN_k[1] + rN_{k+1}[2] = M_k[2] - qN_k[2]N_{k+1}[3] = M_k[3] - qN_k[3]$$

puis $M_{k+1} = N_k$, $N_{k+1}[1] = r = M_k[1] - qN_k[1]$, si bien que :

$$M_{k+1}[2]a + M_{k+1}[3]b = N_k[2]a + N_k[3]b = N_{k+1}[1] = M_{k+1}[1]$$

De même on a :

$$N_{k+1}[2]a + N_{k+1}[3]b = M_k[2]a + M_k[3]b - q(N_k[2]a + N_k[3]b) = M_k[1] - qN_k[1] = N_{k+1}[1]$$

Il est facile de voir qu'à l'instant initial ces deux conditions sont également réalisées.

b L'algorithme d'Euclide étendu implémenté ici a la même structure que l'algorithme d'Euclide : la nombre d'itérations est le même (si on admet que les entiers passés en paramètre sont dans le bon ordre).

Ici, en l'occurrence, on a par propriété du reste dans la division euclidienne, on a $0 \geq N_{k+1} < N_k$, d'où l'arrêt de l'algorithme.

Question 16 *Que renvoie l'algorithme* `egcd` *? Justifier sa validité.*

D'après les deux questions précédentes, en sortie on a $N[1] = 0$ et $M[1] = a \wedge b$, si bien que $M[2]$ et $N[2]$ contiennent une solution de l'équation $ax + by = 1$, par le théorème de Bezout.

Question 17 *Implémenter cette fonction en Caml, de façon récursive, et évaluer sa complexité.*

```
let rec egcd a b =
  if (a mod b) = 0 then (0,1)
  else
    let (x,y) = egcd b (a mod b) in
    (y,x-y*(a/b)) ; ;
```

Comme on l'a expliqué, la seule chose qui change par rapport à un algorithme d'Euclide "classique", c'est le nombre d'opérations faites à chaque passage dans la boucle, ou à chaque appel récursif. On laisse en exercice au lecteur de montrer que l'ordre de grandeur de la complexité de cet algorithme ne change pas pour autant.