Informatique DM, suite d'obtention de puissances : Corrigé

1 Introduction

Question 1. Soit $n \geq 1$. Considérons une suite d'obtention de la puissance n, notée $(n_0, n_1, \ldots, n_r = n)$. Pour $i > 1$ un indice de la suite, la croissance stricte et la condition de sommation impliquent que $n_i \leq 2n_{i-1}$. Une récurrence immédiate montre que $n_r \leq 2^r \Rightarrow r \geq \log_2(n)$. Comme r est un entier, on a $r \geq \lceil \log_2(n) \rceil$. Ainsi, tout calcul de a^n qui n'utilise que des multiplications nécessite un nombre de multiplications au moins égal à $\lceil \log_2(n) \rceil$.

La famille des puissances de deux est une famille infinie de valeurs de n qui peuvent être calculées en effectuant exactement $\log_2(n)$ multiplications : en effet, la suite $(1, 2, \ldots, 2^p)$ de longueur $p+1$ donne un moyen de calculer a^{2^p} avec $p = \log_2(2^p)$ multiplications.

2 Algorithme par division

Question 2. La suite correspondant à l'algorithme *par_ division* est :

- pour l'obtention de la puissance 15, $(1, 2, 3, 6, 7, 14, 15)$, de longueur 7 :
- pour l'obtention de la puissance 16, $(1, 2, 4, 8, 16)$, de longueur 5 ;
- pour l'obtention de la puissance 27, $(1, 2, 3, 6, 12, 13, 26, 27)$, de longueur 8 ;
- pour l'obtention de la puissance 125 ; $(1, 2, 3, 6, 7, 14, 15, 30, 31, 62, 124, 125)$, de longueur 12.

Question 3. Par exemple via usage de la fonction `List.rev` et une fonction auxiliaire.

```
let par_division n =
  let rec aux n=match n with
  | 1 -> [1]
  | _ when n mod 2 = 0 -> n::(aux (n/2))
  | _ -> n::(n-1)::(aux (n/2))
  in List.rev (aux n)
;;
```

Remarque : une fonction auxiliaire avec accumulateur permet de se passer de `List.rev`.

Question 4. Montrons par récurrence sur n que l'algorithme *par_ division* appliqué à n effectue au plus $2 \times \lfloor \log_2(n) \rfloor$ multiplications :

- c'est évident pour $n = 1$, car a^1 s'obtient sans multiplication ;
- soit $n \geq 2$, supposons la propriété vérifiée pour tout entier p vérifiant $1 \leq p < n$. Alors via l'algorithme *par_ division*, a^n s'obtient à partir de $a^{\lfloor n/2 \rfloor}$ via une multiplication (si n pair) ou deux (si n impair) :
  - si n est pair, on a $p = n/2$, d'où $\log_2(n) = 1 + \log_2(p)$. Pour le calcul de a^n il faut donc par principe de récurrence au plus $2 \lfloor \log_2(p) \rfloor + 1 < 2(\lfloor \log_2(p) + 1 \rfloor) = 2 \lfloor \log_2(n) \rfloor$ multiplications. La propriété est vraie au rang n ;
  - si n est impair, on a $n = 2p + 1$, d'où $\log_2(n) > \log_2(p) + 1$, et $\lfloor \log_2(p) + 1 \rfloor \leq \lfloor \log_2(n) \rfloor$. Par principe de récurrence, il faut au plus $2 \lfloor \log_2(p) \rfloor + 2 \leq 2 \lfloor \log_2(n) \rfloor$ multiplications, et là aussi la propriété est vraie au rang n.
- Par principe de récurrence, la propriété est démontrée.

Ce majorant est atteint pour les entiers de la forme $n = 2^p - 1$, $p \geq 1$ par récurrence immédiate : en effet, c'est le cas pour $p = 1$, et pour $p > 1$ on a $\lfloor n/2 \rfloor = 2^{p-1} - 1$.

3 Algorithme par décomposition binaire

Question 5. La suite correspondant à l'algorithme *par_ décomposition_ binaire* est :

- pour l'obtention de la puissance 15, $(1, 2, 3, 4, 7, 8, 15)$, de longueur 7 ;
- pour l'obtention de la puissance 16, $(1, 2, 4, 8, 16)$ de longueur 5 ;
- pour l'obtention de la puissance 27, $(1, 2, 3, 4, 8, 11, 16, 27)$, de longueur 8 ;
- pour l'obtention de la puissance 125, $(1, 2, 4, 5, 8, 13, 16, 29, 32, 61, 64, 125)$, de longueur 12.

Question 6.

```
let rec binaire_inverse n = match n with
  | 0 -> []
  | _ -> (n mod 2)::binaire_inverse (n/2)
;;
```

Question 7. Par exemple :

```
let par_decomposition_binaire n=
  let rec aux acc p2 tot q=match q with
    | [] -> List.rev acc
    | 0::p ->  aux (p2::acc) (2*p2) tot p
    | 1::p when tot <> 0 -> aux ((p2+tot)::p2::acc) (2*p2) (tot+p2) p
    | _::p  -> aux (p2::acc) (2*p2) p2 p
  in aux [] 1 0 (binaire_inverse n)
;;
```

Explications : on suit l'algorithme, `p2` contient la prochaine puissance de 2 à ajouter, `tot` est la valeur associée à la portion des bits de n lus pour le moment. Lorsqu'on lit 0 dans la décomposition de n, il suffit d'ajouter une puissance de 2 à la suite d'obtention de la puissance n, sinon il faut ajouter cette puissance de 2, et la même plus `tot` (sauf dans le cas où `tot` vaut 0).

4 Quelques cas particuliers

Question 8. À partir d'une suite pour l'obtention de la puissance 3^{k-1} (avec $k \geq 1$), on en déduit une pour l'obtention de la puissance 3^k en la prolongeant par $2 \times 3^{k-1}$ et 3^k. Pour l'obtention de la puissance $1 = 3^0$, la suite est de longueur 1, on en déduit donc par récurrence immédiate une suite pour l'obtention de la puissance 3^k de longueur $2k + 1$.

Pour $k = 27 = 3^3$, on en déduit une suite de longueur 7, qui est plus courte que celle fournie par l'algorithme *par_ division* (de longueur 8).

Question 9.

```
let suite3 n=
  let rec aux n = match n with
    | 1 -> [1]
    | _ -> let p=n/3 in let q=aux p in n::(2*p)::q
  in List.rev (aux n) ;;
```

Question 10. On reprend le même raisonnement, il suffit de voir que la puissance 5^k s'obtient ($k \geq 1$) à partir de la puissance $a = 5^{k-1}$ via 3 termes supplémentaires : $2a, 4a, 5a$.

Dans le cas $n = 125 = 5^3$, on obtient donc une suite de longueur 10, plus courte que le résultat obtenu via l'algorithme *par_ division* (de longueur 12).

Question 11. La suite suivante convient : $(1, 2, 3, 6, 9, 15)$. On en déduit que les deux algorithmes *par_ division* et *par_ decomposition_ binaire* ne donnent pas une suite optimale (la plus courte possible), en effet ils donnaient tous deux une suite de longueur 7 pour la puissance 15.

5 Calcul de puissance à partir d'une suite

Question 12. Une version avec boucles `while` et références :

```
let chercher_indice t k=
  let i,j=ref 0, ref 0 and b=ref false in
  while not !b do
    j:= !i ;
    while !j<k && (not !b) do
      if t.( !i) + t.( !j) = t.(k) then b:=true else incr j
    done ;
    incr i
  done ;
  (!i-1, !j)
;;
```

Une version, un peu plus naturelle peut-être, faisant usage d'une exception et de deux boucles `for` :

```
exception Trouve of (int*int) ;;

let chercher_indice t k=
  try
    for i=0 to k-1 do
      for j=i to k-1 do
        if t.(i) + t.(j) = t.(k) then raise (Trouve (i,j))
      done
    done ;
    (0,0)
  with Trouve x -> x
;;
```

Remarque : le résultat `(0,0)` ne sert qu'à assurer la cohérence du type, si `t` est bien un tableau contenant une suite pour l'obtention d'une puissance, l'exception sera levée.

Dans les deux cas, la complexité est $C(k) = O(k^2)$.

Question 13. On utilise un autre tableau pour stocker les puissances calculées :

```
let puissance x t=
  let p = Array.length t in
  let t2=Array.make p x in
  for k=1 to p-1 do
    let i,j=chercher_indice t k in
    t2.(k) <- t2.(i) *. t2.(j)
  done ;
  t2.(p-1)
;;
```

6 Une suite optimale ?

Question 14. Cet algorithme est semblable à l'algorithme de fusion du tri fusion :

```
let rec union q1 q2=match q1, q2 with
  | [],_ -> q2
  | _,[] -> q1
  | x::p, y::q when x=y -> x::(union p q)
  | x::p, y::_ when x>y -> x::(union p q2)
  | _,x::p -> x::(union q1 p)
;;
```

Question 15. On écrit une première fonction permettant de constituer la liste des éléments somme d'un élément `z` et d'un élément `y` de `q`, respectant les conditions. On utilise ensuite cette fonction pour tout `z` de `q`, avec la fonction précédente.

```
let rec somme1 q z n x = match q with
  | [] -> []
  | y::p when y+z<=x -> [] (* inutile de regarder les éléments suivants car q décroissante *)
  | y::p when y+z>n -> somme1 p z n x
  | y::p -> (y+z)::somme1 p z n x
;;
```

```
let rec somme q n x=match q with
  | [] -> []
  | z::p -> union (somme1 q z n x) (somme p n x)
;;
```

Question 16. Voici :

```
let rec suite_optimale_rec n deja_pris possibles = match possibles with
  | [] -> failwith "impossible !"
  | x::q when x=n -> n::deja_pris
  | x::q -> let q1 = suite_optimale_rec n (x::deja_pris) (somme (x::deja_pris) n x) in match q with
    | [] -> q1
    | _ -> let q2=suite_optimale_rec n deja_pris q in if List.length q1<List.length q2 then q1 else q2
;;
```

Question 17.

```
let suite_optimale n = List.rev (suite_optimale_rec n [1] [2]) ;;
```

Question 18. La complexité est probablement exponentielle en n, puisque l'algorithme consiste à examiner toutes les suites pour l'obtention de la puissance n. Même s'il ne semble pas évident de dénombrer toutes ces suites, leur nombre est probablement plus « proche » asymptotiquement du nombre total de suites croissantes de $[\![1, n]\!]$ (qui est 2^n) que d'un polynôme en n.

Remarques : On peut écrire une version plus efficace de `suite_optimale_rec`, qui maintient une référence vers la suite optimale trouvée pour le moment (et une autre vers sa longueur), et « coupe » les appels récursifs dès qu'on dépasse cette longueur. Néanmoins la complexité reste non polynomiale.