Informatique DM : suite d'obtention de puissances

1 Introduction

On rappelle que l'on dispose des deux fonctions `List.length` et `List.rev`, donnant pour la première le nombre d'éléments d'une liste et pour la seconde une liste « miroir » de la première.

On considère un ensemble U muni d'une loi de composition interne associative appelée *multiplication* et possédant un neutre pour cette loi noté e. Cette multiplication est notée avec le signe $\times$.
Par exemple, U peut être l'ensemble des entiers ou des réels munis de la multiplication usuelle, l'élément neutre étant 1. L'ensemble U peut aussi être l'ensemble des matrices carrées (d'entiers ou de réels) d'une même dimension d avec le produit usuel comme multiplication, l'élément neutre étant la matrice identité de dimension d.
Soit $a \in U$ et soit $n \in \mathbb{N}$. On définit a^n de la façon suivante :

- $a^0 = e$
- si $n \geq 1$, $a^n = a^{n-1} \times a$.

La multiplication étant associative, si i et j sont deux entiers positifs ou nuls de somme n alors $a^n = a^i \times a^j$.
Un élément $a \in U$ et un entier $n \in \mathbb{N}^*$ étant donnés, on cherche à calculer a^n en s'intéressant au nombre de multiplications effectuées.
Dans toute la suite, a et n désignent respectivement un élément quelconque de U et un élément de $\mathbb{N}^*$.

Exemple 1 : si $n = 14$, on peut calculer a^{14} en multipliant 13 fois l'élément a par lui même. On effectue alors 13 multiplications.
Exemple 2 : si $n = 14$, on peut calculer a^{14} en calculant a^2 par $a^2 = a \times a$, puis a^3 par $a^3 = a^2 \times a$, puis a^6 par $a^6 = a^3 \times a^3$, puis a^7 par $a^7 = a^6 \times a$, puis enfin a^{14} par $a^{14} = a^7 \times a^7$. On aura ainsi obtenu le résultat en effectuant 5 multiplications.
Exemple 3 : si $n = 14$, on peut aussi calculer a^{14} en calculant a^2 par $a^2 = a \times a$, puis a^4 par $a^4 = a^2 \times a^2$, puis a^6 par $a^6 = a^4 \times a^2$ puis a^8 par $a^8 = a^4 \times a^4$, puis a^{14} par $a^{14} = a^8 \times a^6$. On aura ainsi obtenu le résultat en effectuant 5 multiplications.

L'objectif est de déterminer des algorithmes qui effectuent peu de multiplications. Soit x un nombre réel positif ; on note $\lfloor x \rfloor$ la partie entière par défaut de x et $\lceil x \rceil$ sa partie entière par excès.

On appelle *suite pour l'obtention de la puissance n* toute suite non vide croissante d'entiers distincts $(n_0, \ldots, n_r)$ telle que

- $n_0 = 1$
- $n_r = n$
- pour tout indice k vérifiant $1 \leq k \leq r$, il existe deux entiers i et j distincts ou non vérifiant $0 \leq i \leq k-1$, $0 \leq j \leq k-1$ et $n_k = n_i + n_j$ (la paire $\{i, j\}$ n'est pas forcément unique).

À une suite pour l'obtention de la puissance n correspond une suite de multiplications conduisant au calcul de a^n. Par exemple, la suite $(1, 2, 4, 6, 7, 12, 19)$ correspond au calcul de a^{19} en faisant les 6 multiplications suivantes : $a^2 = a \times a$, $a^4 = a^2 \times a^2$, $a^6 = a^4 \times a^2$, $a^7 = a^6 \times a$, $a^{12} = a^6 \times a^6$, $a^{19} = a^{12} \times a^7$.
Réciproquement, considérons un calcul de a^n dans lequel on fait en sorte d'ordonner les multiplications pour que les puissances calculées soient d'exposants croissants ; on peut associer à ce calcul une suite pour l'obtention de la puissance n.
À l'exemple 1 est associé la suite $(1, 2, 3, 4, 5, \ldots, 14)$ de longueur 14.
À l'exemple 2 est associé la suite $(1, 2, 3, 6, 7, 14)$ de longueur 6.
À l'exemple 3 est associé la suite $(1, 2, 4, 6, 8, 14)$ de longueur 6.
Le nombre de multiplications correspondant à une suite pour l'obtention de la puissance n est égal à la longueur de la suite diminuée de 1.

Question 1. Montrer que tout calcul de a^n qui n'utilise que des multiplications nécessite un nombre de multiplications au moins égal à $\lceil \log_2(n) \rceil$. Donner une famille infinie de valeurs de n qui peuvent être calculées en effectuant exactement ce nombre de multiplications. Justifier la réponse.

2 Algorithme par division

On considère un algorithme appelé *par_ division* ayant pour objectif le calcul de a^n. Cet algorithme s'appuie sur le principe récursif suivant :

si n vaut 1 alors a^n vaut a

sinon

- on calcule la partie entière par défaut, notée q, de $n/2$
- on calcule par l'algorithme *par_ division* la valeur de $b = a^q$
- si n est pair, alors $a^n = b \times b$ et sinon $a^n = (b \times b) \times a$.

Ainsi, pour obtenir a^{14} l'algorithme *par_ division* fait appel au calcul de a^7 qui fait appel au calcul de a^3 (pour obtenir a^6 en multipliant a^3 par a^3 puis a^7 en multipliant a^6 par a) qui fait appel au calcul de a^1 (pour obtenir a^2 puis a^3). Les différentes puissances calculées sont les puissances $1, 2, 3, 6, 7$ et 14. On constate que la suite pour l'obtention de la puissance 14 correspondant à l'algorithme *par_ division* est la suite $(1, 2, 3, 6, 7, 14)$, de longueur 6. De même, la suite pour obtention de la puissance 19 correspondant à l'algorithme *par_ division* est $(1, 2, 4, 8, 9, 18, 19)$ de longueur 7.

Question 2. Calculer (sans justification) la suite correspondant à l'algorithme *par_ division* successivement :

- pour l'obtention de la puissance 15 ;
- pour l'obtention de la puissance 16 ;
- pour l'obtention de la puissance 27 ;
- pour l'obtention de la puissance 125.

Dans chaque cas, indiquer la longueur de la suite obtenue.

Question 3. Ecrire en Caml la fonction `par_division : int -> int list` qui calcule la suite de la puissance n correspondant à l'algorithme *par_ division*.

Question 4. Montrer que l'algorithme *par_ division* appliqué à n effectue au plus $2 \times \lfloor \log_2(n) \rfloor$ multiplications. Montrer que ce nombre est atteint pour un nombre infini de valeurs de n.

3 Algorithme par décomposition binaire

On considère un algorithme *par_ décomposition_ binaire* dont l'objectif est aussi le calcul de a^n. Cet algorithme utilise la décomposition d'un entier suivant les puissances de 2. L'algorithme est expliqué ci-dessous à l'aide d'exemples.

- Soit $n = 14$. On décompose 14 selon les puissances de 2 : $14 = 2 + 4 + 8$. On a donc : $a^{14} = (a^2 \times a^4) \times a^8$, ce qui conduit à calculer les puissances de a d'exposants $2, 4, 8$ mais aussi 6 et 14 ; la suite pour l'obtention de la puissance 14 correspondant à cet algorithme est la suite $(1, 2, 4, 6, 8, 14)$.
- Soit $n = 18$. On a $18 = 2 + 16$ et donc $a^{18} = a^2 a^{16}$. L'algorithme calcule les puissances d'exposant $2, 4, 8, 16$ puis 18 ; la suite pour l'obtention de la puissance 18 correspondant à cet algorithme est la suite $(1, 2, 4, 8, 16, 18)$.
- Soit $n = 101$. On a $101 = 1 + 4 + 32 + 64$. L'algorithme calcule a^{101} en utilisant les multiplications impliquées par la formule $a^{101} = ((a \times a^4) \times a^{32}) \times a^{64}$; on calcule les puissances $2, 4, 5$ (pour $a \times a^4 = a^5$), $8, 16, 32, 37$ (pour $a^5 \times a^{32} = a^{37}$), 64 et 101 (pour $a^{37} \times a^{64} = a^{101}$) ; la suite pour l'obtention de la puissance 101 correspondant à cet algorithme est la suite $(1, 2, 4, 5, 8, 16, 32, 37, 64, 101)$.

De manière générale, l'algorithme procède en écrivant la décomposition unique de n comme une somme de puissances croissantes du nombre 2, et calcule la valeur cible de a^n en effectuant les produits correspondant aux sommes partielles de cette somme.

Question 5. Calculer (sans justification) la suite correspondant à l'algorithme *par_ décomposition_ binaire* successivement :

- pour l'obtention de la puissance 15 ;
- pour l'obtention de la puissance 16 ;
- pour l'obtention de la puissance 27 ;
- pour l'obtention de la puissance 125.

Dans chaque cas, indiquer la longueur de la suite obtenue.

On considère la décomposition de n suivant les puissances croissantes du nombre 2 :

$$n = c_0 + c_1 \times 2 + \cdots + c_i \times 2^i + \cdots + c_k \times 2^k$$

où pour i vérifiant $0 \leq i < k$, le coefficient c_i vaut 0 ou 1 et c_k vaut 1.
On appelle *écriture binaire inverse* de n la suite $(c_0, c_1, \ldots, c_k)$.
Par exemple, l'écriture binaire inverse de l'entier 14 est $(0, 1, 1, 1)$, celle de l'entier 18 est $(0, 1, 0, 0, 1)$ et celle de l'entier 101 est $(1, 0, 1, 0, 0, 1, 1)$.

Question 6. Ecrire en Caml une fonction `binaire_inverse : int -> int list` qui à un entier $n \geq 1$ donné associe une liste correspondant à son écriture binaire inverse.

Question 7. Ecrire en Caml la fonction `par_decomposition_binaire : int -> int list` qui à un entier $n \geq 1$ donné associe une liste correspondant à l'algorithme *par_ decomposition_ binaire*.

4 Quelques cas particuliers

Question 8. On suppose que $n = 3^k$ où $k \in \mathbb{N}$. En utilisant la formule $3^k = 3^{k-1} + 2 \times 3^{k-1}$, montrer qu'il existe une suite pour l'obtention de la puissance n de longueur $2k + 1$. Indiquer la longueur de cette suite pour $n = 27$ et comparer avec le résultat obtenu par l'algorithme *par_ division*.

Question 9. Soit $k \in \mathbb{N}$. Ecrire en Caml une fonction `suite_3` qui à un entier n supposé être une puissance de 3 associe la liste de longueur $2k + 1$ évoquée en question précédente.

Question 10. On suppose que $n = 5^k$ avec $k \in \mathbb{N}$. Montrer qu'il existe une suite pour l'obtention de la puissance n de longueur $3k + 1$. Indiquer la suite dans le cas $n = 125$ et comparer avec le résultat dans le cas de l'algorithme *par_ division*.

Question 11. Donner une suite de longueur 6 pour l'obtention de la puissance 15. Qu'en déduire quant aux algorithmes *par_ division* et *par_ decomposition_ binaire* étudiés précédemment ?

5 Calcul de puissance à partir d'une suite

Question 12. On considère un tableau `t`, indicé à partir de 0, contenant une suite pour l'obtention d'une certaine puissance positive n. Soit k un entier compris entre 1 et la longueur de `t` diminuée de 1. Soit *val* la valeur contenue dans `t` à l'indice k. On sait que *val* est la somme de deux valeurs du tableau `t` situées à des indices (éventuellement confondus, éventuellement non uniques) strictement inférieurs à k. Programmer une fonction `chercher_indice : int array -> int -> int*int` qui étant donnés `t` et k calcule un couple (i, j) convenable (n'importe lequel). On supposera que k vérifie les contraintes exposées ci-dessus.

Par exemple, si `t = [|1;2;3;4;7;14;17;31|]`, la longueur du tableau est 8. Si $k = 6$, *val* vaut 17 et la fonction renvoie $(2, 5)$ (correspondant aux valeurs 3 et 14 du tableau). Si $k = 1$, la fonction renvoie $(0, 0)$.

Indiquer (sans justification) la complexité $C(k)$ de cette fonction.

Question 13. Soit x un réel représenté par un élément de type `float`. Soit `t` un tableau contenant une suite pour l'obtention d'une certaine puissance positive n. Ecrire en Caml une fonction `puissance : float -> int array -> float` qui prend en argument x et `t` et renvoie x^n en utilisant la tactique associée à la suite associée à `t`.

Contrainte : la complexité de la fonction `puissance` devra être en $O(h \times C(h))$ (ce que l'on ne demande pas de justifier), avec C la complexité précédente et h la longueur de `t`. Rappel : `*.` pour la multiplication de deux flottants.

6 Une suite optimale ?

On propose ici un algorithme *suite_ optimale_ rec* permettant de calculer une suite de longueur minimale pour l'obtention de la puissance n, qu'on supposera au moins égale à 2.

- l'algorithme, récursif, fait usage de deux listes : les éléments *deja_ pris*, et les éléments *possibles*. Initialement, *deja_ pris* contient uniquement 1, et *possibles* ne contient que 2. Ces deux listes sont triées dans l'ordre *strictement décroissant*, et les éléments de *possibles* sont constitués d'entiers strictement supérieurs à ceux de *deja_ pris*, et inférieurs ou égaux à n, que l'on peut obtenir en faisant la somme de deux éléments (éventuellement égaux) de *deja_ pris* ;

- Un appel à *suite_optimale_rec* doit compléter de manière optimale les éléments de *deja_pris* via au moins un élément de *possibles* (et d'autres, calculés ensuite) pour former une suite de longueur minimale pour l'obtention de n, construite dans l'ordre décroissant. Lors d'un appel, il y a plusieurs possibilités :
  - soit le premier élément de *possibles* est n : la suite optimale est simplement constituée de n et des éléments de *deja_pris*.
  - sinon, on peut choisir de prendre ou non le premier élément de *possibles*. Il est donc nécessaire de faire deux appels récursifs (si *possibles* contient au moins deux éléments) : dans le premier appel récursif, le premier élément de *possibles* (le plus grand) est transféré dans *deja_pris*, et il est nécessaire de recalculer *possibles*. Dans le second (si *possibles* contenait au moins deux éléments), on a simplement supprimé le premier élément de *possibles*. Après les deux appels récursifs (s'il y en a bien eu deux), les longueurs des suites obtenues sont comparées, et la plus grande est renvoyée.

Les questions qui suivent implémentent l'algorithme *suite_optimale_rec* sous la forme d'une fonction de signature `int -> int list -> int list -> int list`, l'appel `suite_optimale_rec n deja_pris possibles` devant suivre le principe précédent.

Question 14. Écrire une fonction `union` de type `int list -> int list -> int list` prenant en entrée deux listes triées dans l'ordre décroissant, chacune constituée d'éléments distincts, et renvoyant la liste triée dans l'ordre décroissant des éléments se trouvant dans l'une ou dans l'autre, sans doublon.

Question 15. À l'aide de la fonction précédente, écrire une fonction `somme q n x`, ayant pour signature `int list -> int -> int -> int list`, prenant en entrée une liste non vide triée dans l'ordre décroissant, deux entiers $n > x$, et renvoyant une liste triée dans l'ordre décroissant, constituée d'éléments distincts y, qui sont somme de deux éléments de `q` (éventuellement égaux), et tels que $x < y \leq n$.

Question 16. Implémenter la fonction `suite_optimale_rec n deja_pris possibles`.

Question 17. En déduire une fonction `suite_optimale n` renvoyant une suite de longueur minimale pour l'obtention de la puissance n, dans l'ordre croissant.

Question 18. Que pensez-vous de la complexité de cet algorithme ? Argumenter.