Travaux pratiques 4

Partitions

Option informatique MPSI1 & MPSI2

Après un casse réussi, deux cambrioleurs veulent partager équitablement leur butin. Ils commencent par s'accorder sur la valeur de chacun des n objets volés ; ceci leur donne un ensemble $X = \{x_0, \dots, x_{n-1}\}$ de nombres entiers. Ils leur faut alors résoudre le problème suivant : déterminer une partition de X en deux sous-ensembles Y et Z de même somme. Nos deux malfaiteurs sont confrontés au **problème de partition** : nous dirons que X est une **instance** de ce problème.
Le **problème de décision** associé s'énonce ainsi : une telle partition existe-t-elle ?
On peut généraliser le problème de décision en problème de somme partielle (SSP, Subset Sum Problem en anglais) : étant donnée une suite $X = (x_0, x_1, \dots, x_{n-1})$ d'entiers positifs et un entier k existe-t-il un sous ensemble $I \subset \{0, 1, \dots, n-1\}$ tel que $\sum_{i \in I} x_i = k$?

Si on $\|X\|$ la somme des éléments de X : $\|X\| = \sum_{i=0}^{n-1} x_i$ le problème de décision de partition est le problème de somme partielle pour $k = \frac{\|X\|}{2}$, avec $\|X\|$ pair.

Dans ce TP, la suite X sera représentée par une liste.
Pour résoudre le problème (de décision) SSP on cherchera une fonction

```
f : int list -> int -> bool
```

telle que `f x k` renvoie `true` s'il existe une partie de X, représenté par `x`, dont la somme est k et qui renvoie `false` sinon.
On demandera aussi une fonction

```
f_sol : int list -> int -> int list
```

telle que `f_sol x k` renvoie une liste `x1` extraite de `x` dont la somme est k et qui renvoie `[-1]` sinon. On dira que `f_sol` choisit une solution.

I Passage en force

Nous allons commencer par traduire l'énoncé directement. On doit donc

- déterminer toutes les parties $I \subset \{0, 1, \dots, n-1\}$,
- calculer $\sum_{i \in I} x_i$ tant que cette somme est distincte de k

Une idée (itérative) est de remarquer qu'on peut caractériser une partie de $\{0, 1, \ldots, n-1\}$ par la suite des valeurs (0 ou 1) de sa fonction caractéristique et qu'une telle suite de 0 et 1 peut être vue comme la suite issue de la décomposition en base 2 d'un entier entre 0 et $2^n - 1$.
On peut ainsi écrire la somme partielle associée à un entier p :

```
let rec somme_partielle liste p =
   match liste with
   |[] -> 0
   |t::q -> t*(p mod 2) + somme_partielle q (p/2);;
```

Exercice 1
Que donne `somme_partielle [4; 2; 1; 2; 3] 25` ?
Déterminer p pour que `somme_partielle [a; b; c; d; e] p` renvoie $a + b + d$.

Exercice 2
Écrire une fonction `puissance2 : int list -> int` qui renvoie 2^n où n est la longueur de la liste **sans** utiliser `List.length` et en moins de 5 lignes.

Exercice 3
Écrire une fonction `brute_force` qui utilise la méthode exposée pour résoudre SSP.

Exercice 4
Calculer le nombre d'opérations élémentaires effectuées dans le pire des cas (quand la réponse est `false` en fonction de la taille n de la liste.

Exercice 5
Modifier la fonction `brute_force` en une fonction `brute_force_sol` qui choisit une solution.

II Introduction de la récursivité

Une autre méthode pour déterminer les listes extraites d'une liste est de remarquer que les listes extraites de `t::q` sont

- soit des listes extraites de q
- soit des listes de la forme `t::q1` avec `q1` extraite de q.

Ici on traduit le problème par k est une somme extraite de `t::q` si et seulement si

- k est une somme extraite de `q` ou
- $k - t$ est une somme extraite de `q`.

Exercice 6
Écrire une fonction `recursivite` qui utilise la méthode exposée pour résoudre SSP.

Exercice 7
Calculer le nombre d'opérations élémentaires effectuées dans le pire des cas (quand la réponse est `false` en fonction de la taille n de la liste.

Exercice 8
Comme les termes sont positifs on peut écrire un algorithme dont espère qu'il sera plus rapide en utilisant le fait que la réponse est immédiate quand $k = 0$ ou $k < 0$.
Modifier `recursivite` dans ce sens.

Exercice 9
Modifier la fonction `recursivite` en une fonction `recursivite_sol` qui choisit une solution.

III Programmation dynamique

La formule ; k est une somme extraite de `t::q` si et seulement si

- k est une somme extraite de `q` ou
- $k - t$ est une somme extraite de `q`.

permet d'exprimer le résultat pour une liste de taille n et un entier k à partir de deux résultats pour le reste de la liste et des entiers inférieurs à k : des sous-problèmes. Les conditions d'un algorithme utilisant la programmation dynamique sont réunies.
Si la suite des éléments est $X = (x_0, x_1, \ldots, x_{n-1})$ et que l'objectif est k, on définit une matrice de booléens `m` de taille $(n+1) \times (k+1)$ telle que `m.(i).(j)` vaut `true` si et seulement si j est une somme partielle de $X_i = (x_0, x_1, \ldots, x_{i-1})$.

Exercice 10
Écrire une fonction `dynamique` qui utilise la méthode de programmation dynamique pour résoudre SSP. On pourra utiliser la fonction `Array.of_list` pour convertir une liste en tableau mais cela n'est pas obligatoire.

Exercice 11
Déterminer la complexité de votre fonction en fonction de n et de k.
Donner un exemple pour lequel $k \leqslant \|X\|$ et la complexité n'est pas polynomiale en n.

Exercice 12
La complexité spatiale, nk, peut être importante. Écrire une une fonction `dynamique1` qui utilise uniquement un tableau de taille $k + 1$ pour résoudre SSP.

Exercice 13
Écrire une fonction `dynamique_sol` qui choisit une solution.

Solutions

Solution de l'exercice 1
On obtient $1.4 + 0.2 + 0.1 + 1.2 + 1.3 = 9$.
On doit prendre $p = 1.2^0 + 1.2^1 + 0.2^2 + 1.2^3 + 0.2^4 = 11$.

Solution de l'exercice 2

```
let rec puissance2 liste =
   match liste with
   |[] -> 1
   |t::q -> 2*(puissance2 q);;
```

Solution de l'exercice 3

```
let brute_force liste k =
   let reponse = ref false in
   let i = ref 0 in
   let max = puissance2 liste in
   while not !reponse && !i < max do
      if somme_partielle liste !i = k
      then reponse := true;
      incr i done;
   !reponse;;
```

Solution de l'exercice 4
`puissance2` effectue n multiplications.
`somme_partielle` effectue 4 opérations pour chaque élément de la liste donc $4n$ en tout.
La complexité est donc $n + 4n.2^n = \mathcal{O}(n.2^n)$.

Solution de l'exercice 5
On commence par écrire l'extraction d'une liste à l'aide d'un entier qui va représenter la fonction indicatrice.

```
let rec extraire liste p =
   match liste with
   |[] -> []
   |t::q when p mod 2 = 0 -> extraire q (p/2)
   |t::q -> t :: (extraire q (p/2));;
```

```
let brute_force_sol liste k =
   let sol = ref (-1) in
   let i = ref 0 in
   let max = puissance2 liste in
   while !sol = -1 && !i < max do
      if somme_partielle liste !i = k
      then begin sol := !i;
      incr i done;
   if !sol = -1
   then [-1]
   else extraire liste !sol;;
```

Solution de l'exercice 6

```
let rec recursivite liste k =
   match liste with
   |[] -> k = 0
   |t::q -> recursivite q k || recursivite q (k-t);;
```

Solution de l'exercice 7

On note $C(n)$ la complexité maximale pour une liste de taille n.
$C(0) = 1$ (la comparaison)
$C(n) = 2 + 2C(n-1)$ car on calcule un "ou" et une différence.
En posant $u_n = C(n) + 2$ on a $u_0 = 3$ et $u_n = 2u_{n-1}$ donc $u_n = 3.2^n$ puis $C(n) = 3.2^n - 2$.

Solution de l'exercice 8

```
let rec recursivite liste k =
   match liste, k with
   |_, 0 -> true
   |[], _ -> false
   |t::q, k when k < t -> recursivite q k
   |t::q, _ -> recursivite q k || recursivite q (k-t);;
```

Solution de l'exercice 9

```
let rec recursivite_sol liste k =
   match liste, k with
   |_, 0 -> []
   |[], _ -> [-1]
   |t::q, k when k < t -> recursivite_sol q k
   |t::q, _ -> begin match recursivite_sol q (k-t) with
                     |[-1] ->  recursivite_sol q k
                     |l -> t::l end;;
```

Solution de l'exercice 10

La formule sur les sous-problèmes n'est valide que pour $t \leqslant k$.
Si on convertit en tableau.

```
let dynamique liste k =
   let tab = Array.of_list liste in
   let n = Array.length tab in
   let m = Array.make_matrix (n+1) (k+1) false in
   m.(0).(0) <- true;
   for i = 1 to n do
      for j = 0 to k do m.(i).(j) <- m.(i-1).(j) done;
      let t = tab.(i-1) in
      for j = t to k do m.(i).(j) <- m.(i).(j) || m.(i-1).(j-t)
         done done;
   m.(n).(k);;
```

Sans conversion.

```
let dynamique liste k =
   let n = List.length liste in
   let m = Array.make_matrix (n+1) (k+1) false in
   m.(0).(0) <- true;
   let rec aux reste i =
      match reste with
      |[] -> m.(n).(k)
      |t::q -> for j = 0 to k do m.(i).(j) <- m.(i-1).(j) done;
               for j = t to k do m.(i).(j) <- m.(i).(j) || m.(i-1)
                  .(j-t) done;
               aux q (i+1) in
   aux liste 1;;
```

Solution de l'exercice 11
La complexité est un $\mathcal{O}(nk)$.
Si on a $x_i = 3^i$ on a $\|X\| = \frac{1}{2}(3^n - 1)$ donc, pour $k = \frac{3}{4}.3^n$, la complexité sera de l'ordre de $n.3^n$.

Solution de l'exercice 12

```
let dynamique1 liste k =
   let tab = Array.of_list liste in
   let n = Array.length tab in
   let m = Array.make (k+1) false in
   m.(0) <- true;
   for i = 1 to n do
      let t = tab.(i-1) in
      for j =  k downto t do m.(j) <- m.(j) || m.(j-t) done done;
   m.(k);;
```

Solution de l'exercice 13
On va utiliser un tableau de solutions.

```
let dynamique_sol liste k =
   let tab = Array.of_list liste in
   let n = Array.length tab in
   let sol = Array.make (k+1) [-1] in
   m.(0) <- [];
   for i = 1 to n do
      let t = tab.(i-1) in
      for j =  k downto t do
         if sol.(j-t) <> [-1]
         then sol.(j) <- t::sol.(j-t) done done;
   sol.(k);;
```