TD 11 : Programmation Dynamique

1 Partition et Subset Sum

La question de base qui nous intéresse pour cet exercice est la suivante : étant donné une liste qui contient des entiers positifs, est-ce qu'on peut la partitionner en deux parties qui ont la même somme ? Par exemple, pour la liste $[1, 2, 3, 4]$ la réponse et oui, car on peut la partitionner en deux sous–listes $[1, 4]$ et $[2, 3]$ qui ont la même somme. De la même façon, une solution pour la liste $[1, 5, 7, 13]$ est la partition $[1, 5, 7]$, $[13]$ (les deux parties n'ont pas forcement le même nombre d'éléments, seulement la même somme). Pour la liste $[1, 2, 3, 4, 5]$ la réponse est évidemment qu'une telle partition n'existe pas (car la somme de tous les éléments est impaire).

1.1 Solution Récursive Exponentielle

Donner une fonction `partition :: [Int] -> Bool` qui, étant donné une liste, décide si une partition en deux listes de sommes égales existe. Donner aussi une deuxième version de votre fonction avec la signature `partition2 :: [Int] -> Maybe ([Int],[Int])` qui retourne la partition si elle existe (ou `Nothing` sinon).

La façon la plus simple de programmer une telle fonction est de générer toutes les partitions possibles de la liste donnée et puis tester chacune pour voir si les sommes des deux parties sont égales.

Exemples d'utilisation :

```
*Main> partition [1..8]
True
*Main> partition2 [1..8]
Just ([3,7,8],[1,2,4,5,6])
*Main> partition2 [1,4..35]
Just ([1,4,7,28,31,34],[10,13,16,19,22,25])
*Main> partition2 [1,2,4,8,16]
Nothing
*Main> partition2 [1,2,4,8,15]
Just ([15],[1,2,4,8])
```

1.2 Subset Sum

La fonction `partition` qu'on vient d'implémenter a une complexité **exponentielle** d'environ 2^n, où n est la taille de la liste donnée, car on considère (potentiellement) toutes les partitions possibles. Cette complexité rend la fonction très rapidement **inutilisable**, même si on considère des listes de quelques dizaines d'éléments. Notre objectif dans cet exercice est d'utiliser les principes de la programmation dynamique pour arriver à une complexité polynomiale en tant que fonction de n.

Notre première étape est (paradoxalement) de formuler un problème qui est **plus difficile** que le problème donné. Considérez la question suivante : vous avez une liste `xs` et un entier `s` et on vous demande si il existe un sous-ensemble de `xs` dont la somme est exactement `s`. On appelle ce problème Subset Sum.

Pourquoi dit–on que ce problème est plus difficile que le problème de la partition ?

– **Réponse :** Car, `partition xs` doit retourner `True` si et seulement si `sum xs` est paire et on peut trouver un sous-ensemble de `xs` de somme `sum xs `div` 2`. Donc, si on arrive à résoudre Subset Sum,

on obtient automatiquement un algorithme pour le problème de partition, ce qui montre que Subset Sum est un problème plus général.

Programmer une fonction `subsetsum :: [Int] -> Int -> Bool` qui, étant donné `xs, s` retourne `True` si un sous-ensemble de `xs` de somme `s` existe. En plus, programmer une deuxième version de la même fonction avec signature `[Int] -> Int -> Maybe [Int]` qui retourne le sous-ensemble s'il existe.

Vous pouvez ici donner une implémentation récursive directe, sans utiliser la programmation dynamique. Les idées principales sont : pour le cas de base, si `s==0` ou `xs==[]` c'est facile de retourner la bonne réponse ; pour le cas général, si `(x:xs)` est la liste donnée, soit on ignore l'élément `x` (et donc on cherche un sous-ensemble de `xs` de somme `s`), soit `x` fait partie de la solution (et donc on cherche un sous-ensemble de `xs` de quelle somme ?)

Exemples d'utilisation :

```
*Main> subsetsum [1..10] 15
True
*Main> subsetsum2 [1..10] 15
Just [7,8]
*Main> subsetsum2 [1..10] 35
Just [5,6,7,8,9]
*Main> subsetsum2 [1..20] (sum [1..20])
Just [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
(0.90 secs, 402,779,496 bytes)
*Main> subsetsum2 [1..22] (sum [1..22])
Just [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22]
(3.22 secs, 1,610,741,144 bytes)
*Main> subsetsum2 [1..24] (sum [1..24])
Just [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24]
(12.35 secs, 6,442,589,432 bytes)
*Main> subsetsum2 [1..25] (sum [1..25])
Just [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25]
(24.28 secs, 12,885,043,888 bytes)
```

Notez bien l'augmentation très rapide (exponentielle) du temps d'exécution !

1.3 Subset Sum Malin

Enfin, dans cette partie on s'attaque au problème du Subset Sum en utilisant la programmation dynamique avec l'objectif d'éviter la complexité de 2^n. Pour rappel, le principe de la programmation dynamique est qu'on sauvegarde des valeurs intermédiaires qui représentent les solutions de sous–problèmes de notre problème, afin d'éviter de les recalculer. À première vue, voyons pourquoi ce principe pourrait nous est utile. Considérons l'exemple où on est donné la liste $[1, 2, 3, 4, 5, 6]$ et $s = 20$. L'algorithme de la question précédente considère deux branches : pour chaque élément x, soit on l'écarte (et on tente de trouver la somme s en utilisant les éléments qui restent), soit on le garde (et on tente de former la somme $s - x$ à partir des autres). Donc, une branche de cette algorithme va, par exemple, sélectionner $1, 2$ et écarter 3. Une autre, va écarter $1, 2$ et sélectionner 3. L'observation clé est maintenant que ces deux branches sont équivalentes : les deux vont tenter de former la somme $20 - 3 = 20 - 1 - 2 = 17$ à partir de la liste $[4, 5, 6]$. Donc, ça n'a pas de sens de les calculer toutes les deux !

Plus formellement maintenant, l'idée de l'algorithme de programmation dynamique qu'on va tenter de programmer est la suivante : on va construire une matrice $S[i, j]$ de n lignes et s colonnes. L'élément $S[i, j]$ va contenir une solution qui (i) utilise un sous-ensemble des i premiers éléments de la liste (ii) et donne la somme j. Avec cette formulation, $S[n, s]$ contient la solution souhaitée. Il nous suffit donc de formuler des cas de base (comment calculer $S[0, j]$?), et le cas général qui calcule $S[i, j]$ à partir de la ligne $(i - 1)$ de la matrice.

Programmer une fonction `subsetsum3` qui fonctionne de cette façon. Votre fonction devrait avoir une complexité polynomial en n et s. En utilisant cette fonction donner une fonction `fastpartition` qui améliore la complexité de la fonction de la première question.

```
*Main> subsetsum3 [1..50] (sum [1..50])
Just [50,49,48,47,46,45,44,43,42,41,40,39,38,37,36,35,34,33,32,31,30,29,28,
   27,26,25,24,23,22,21,20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1]
(0.74 secs, 49,585,848 bytes)
*Main> subsetsum3 [1..80] (sum [1..80])
Just [80,79,78,77,76,75,74,73,72,71,70,69,68,67,66,65,64,63,62,61,60,59,58,
   57,56,55,54,53,52,51,50,49,48,47,46,45,44,43,42,41,40,39,38,37,36,35,34,
   33,32,31,30,29,28,27,26,25,24,23,22,21,20,19,18,17,16,15,14,13,12,11,10,
   9,8,7,6,5,4,3,2,1]
(13.88 secs, 207,104,520 bytes)
*Main> l = map (\x -> x^2) [1..35]
(0.01 secs, 0 bytes)
*Main> fastpartition l
Just ([784,729,676,625,576,529,484,441,400,361,324,289,256,196,169,144,121,
   100,81,64,49,36,16,4,1],[9,25,225,841,900,961,1024,1089,1156,1225])
(13.47 secs, 148,330,184 bytes)
```

Solution :

```haskell
-- Partition in 2^n time

partition :: [Int] -> Bool
partition xs = [] /= (filter (\(xs,ys)-> sum xs == sum ys) $ allparts xs)
 where
  allparts [x] = [([x],[])]
  allparts (x:xs) = concatMap (\(l,r)-> [(x:l,r),(l,x:r)]) $ allparts xs
--  Equivalently, we can say the following:
--  allparts (x:xs) =  allparts xs >>= (\(l,r)-> [(x:l,r),(l,x:r)])

allparts :: [a] -> [([a],[a])]
allparts [x] = [([x],[])]
allparts (x:xs) = concatMap (\(l,r)-> [(x:l,r),(l,x:r)]) $ allparts xs


partition2 :: [Int] -> Maybe ([Int],[Int])
partition2 xs
 | [] /= sols = Just $ head sols
 | otherwise = Nothing
 where
  allparts [x] = [([x],[])]
--  allparts (x:xs) = concatMap (\(l,r)-> [(x:l,r),(l,x:r)]) $ allparts xs
  allparts (x:xs) =  allparts xs >>= (\(l,r)-> [(x:l,r),(l,x:r)])
  sols = (filter (\(xs,ys)-> sum xs == sum ys) $ allparts xs)

-- Subset Sum in 2^n time

subsetsum :: [Int] -> Int -> Bool
subsetsum _ 0 = True
subsetsum [] _ = False
```

```haskell
subsetsum (x:xs) s = subsetsum xs s || subsetsum xs (s-x)

subsetsum2 :: [Int] -> Int -> Maybe [Int]
subsetsum2 _ 0 = Just []
subsetsum2 [] _ = Nothing
subsetsum2 (x:xs) s
 | isJust sol1 = sol1
 | otherwise = (x:) <$> sol2
 where
  isJust (Just _) = True
  isJust _ = False
  sol1 = subsetsum2 xs s
  sol2 = subsetsum2 xs (s-x)

-- Subset Sum DP pseudo-polynomial time

subsetsum3 :: [Int] -> Int -> Maybe [Int]
subsetsum3 xs s = ss3 (length xs) s
 where
  ss3 _ 0 = Just []
  ss3 0 _ = Nothing
  ss3 i s
   | isJust (sols !! (i-1) !! s) = sols !! (i-1) !! s
   | s<xs!!(i-1) = Nothing
   | otherwise = ((xs!!(i-1)):) <$> sols !! (i-1) !! (s - (xs!!(i-1)))
  sols = [ [ ss3 i j  | j<-[0..s] ]   | i<-[0..length xs] ]
  isJust (Just _) = True
  isJust _ = False

fastpartition :: [Int] -> Maybe ([Int],[Int])
fastpartition xs
 | sum xs `mod` 2 == 1 = Nothing
 | otherwise = (\l-> (l,compl xs l)) <$> sol
 where
  sol = subsetsum3 xs (sum xs`div`2)
  compl xs l = [ x | x<-xs, not (elem x l) ]
```