info421 : Programmation fonctionnelle
TP 4 : algorithmes gloutons, mémoization

Responsables : Pierre Hyvernat et Krzysztof Worytkiewicz
Laboratoire de mathématiques de l'université de Savoie
email : Pierre.Hyvernat@... / Krzysztof.Worytkiewicz@... (...@univ-savoie.fr)
http://lama.univ-savoie.fr/~hyvernat/
http://lama.univ-savoie.fr/~worytkiewicz/

Partie 1 : Mémoization

Question 1. Vous aimeriez optimiser le calcul des valeurs d'une fonction `f` de la manière suivante : lors du calcul de `f x`, vous regardez si la valeur correspondante a déjà été calculée.
- Si oui, vous renvoyez la valeur déjà calculée
- si non, vous calculez `f x` normalement, et conservez la valeur quelque part.

Cette technique s'appelle "mémoization". Le deuxième calcul de `f x` est en général plus rapide que le premier...

Utilisez cette méthode pour modifier la définition

```
let rec fib1 n =
    if n < 2
    then n
    else fib1 (n-1) + fib1 (n-2)
```

Pour stocker les résultats, vous pouvez utiliser une liste d'association qui contient les paires `(n, v)` où `v` est la valeur de `fib n`. La fonction `List.assoc : 'a -> ('a*'b) list -> 'b` permet de récupérer une valeur `v` à partir de la valeur `n` correspondante.

Attention, "`List.assoc n table`" lève l'exception "`Not_found`" lorsque "`n`" n'apparait pas dans "`table`".

Partie 2 : Rendre de la monnaie

La boulangère dispose d'une quantité illimitée de pièces de 1, 2, 5 et 10 centimes. Elle peut obtenir n'importe quelle somme à partir de ces pièces, mais comment faire pour minimiser le nombre total de pièces utilisées ?

Question 1. Proposez un algorithme simple pour trouver une solution optimale. Programmez une fonction correspondante :

```
monnaie_glouton : int list -> int -> (int*int) list
```

devra prendre en argument la liste des pièces qu'on a le droit d'utiliser et le montant total qu'on veut obtenir. Le résultat sera une solution qui minimise le nombre total de pièces utilisées.

Par exemple : `monnaie_glouton [1 ; 2 ; 5 ; 10] 9` donnera `[ (5,1) ; (2,2) ]` car il suffit d'une pièce de 5 centimes et de 2 pièces de 2 centimes pour faire 9 centimes.

Question 2. Sur la planète Nethack, les habitants utilisent des pièces de 1, 4 et 6 zorkmid. Faites fonctionner votre fonction sur les entiers de 1 à 10. Que constatez-vous ?

Question 3. Pour corriger le problème de la question précédente, nous allons utiliser un algorithme récursif plus complexe. Nous allons commencer par seulement calculer le nombre total de pièces utilisées :

```
compte_monnaie 0 _ = 0
compte_monnaie _ [] = "+infini"
```

```
compte_monnaie n (p::l) = min (compte_monnaie n l)
                              (1 + (compte_monnaie (n-p) (p::l)))
```

Le premier cas correspond au cas où on n'utilise pas la pièce `p`, et le second au cas où on utilise la pièce `p`.

Écrivez la fonction `monnaie_rec` correspondante.

Attention : si vous modélisez "`+infini`" par "`max_int`" (le plus grand entier connu par Caml), vous aurez que "`max_int + 1 = min_int`" ! Pour éviter le problème, vous pouvez utiliser un type `option` avec

- "`n`" représenté par "`Some n`",
- "`+infini`" représenté par "`None`".

(Mais dans ce cas, il faudra réécrire les fonctions "`min`" et "`1+...`".)

Question 4. Écrivez une version mémoizée de cette fonction.

Question 5. Estimez la complexité, en fonction de `n` des fonctions

- `monnaie_glouton`,
- `compte_monnaie`,
- `monnaie_memoizee`.

(Essayez juste de donner un ordre de grandeur...)

Question 6. Généralisez ce principe pour obtenir le nombre de chacune des pièces, et écrivez la fonction correspondante :

```
monnaie_rec : int list -> int -> (int*int) list
```