Éléments de correction du devoir surveillé n° 3.

I Petit exercice : un type utile pour les enseignants...

Correction dans le fichier `TypeNote.ml` .

II Problème : méthode diviser pour régner appliquée au calcul des termes de la suite de Fibonnaci

Partie A. Calcul naïf de F_n

1. Cette fonction fait deux appels récursifs avec des paramètres **à peine plus petits** que le paramètre reçu. Cela se traduit par une complexité exponentielle, inexploitable en pratique même pour des valeurs assez petites de n. La relation de récurrence permettant de calculer le coût $C(n)$ est $C(n) = C(n-1) + C(n-2) + O(1)$.
2. La fonction `fibo` présente deux cas de base dont l'un est l'élément minimal de $\mathbb{N}$, qui muni de l'ordre naturel sur les entiers est un ensemble bien fondé. Dans le cas d'induction, la fonction fait un nombre d'appels fini de la fonction avec des paramètres valides (positifs) strictement plus petits. La fonction va donc bien terminer.

Partie B. Première amélioration

1. Le code est assez simple. Par exemple (fnm1 veut dire "F n moins 1", soit F_{n-1} par exemple) :

```
let rec fibo_aux (n : int) : int * int =
  match n with
  | 1 -> (0, 1)
  | _ -> let (fnm2, fnm1) = fibo_aux (n - 1) in (fnm1, fnm1 +
   fnm2) ;;
```

2. D'où la fonction encapsulant l'appel à la fonction précédente :

```
let fibo_bis (n : int) : int =
  match n with
  | 0 -> 0
  | _ -> snd (fibo_aux n);;
```

3. La complexité est maintenant linéaire $\mathcal{O}(n)$, ce qui se prouve très simplement. Le coût de `fibo_bis` est clairement le même que celui de de `fibo_aux` .

   Pour `fibo_aux` , la relation de récurrence pour calculer le coût est simplement de la forme $C(n) = C(n-1) + a$ où a est une constante. Par sommation et télescopage il reste une complexité linéaire.

Partie C. Encore mieux !

1. et

2. D'après les indication de l'énoncé il vient facilement : $T_{2n} = (F_{2n-1}, F_{2n}) = \left(F_n^2 + F_{n-1}^2, F_n^2 + 2F_nF_{n-1}\right)$ et $T_{2n+1} = (F_{2n}, F_{2n+1}) = \left(F_n^2 + 2F_nF_{n-1}, 2F_n^2 + F_{n-1}^2 + 2F_nF_{n-1}\right)$, ce qui montre bien que l'on peut exprimer T_{2n} et T_{2n+1} à partir de T_{2n}.
3. Hormis les cas de base, pour calculer F_n on va essentiellement calculer T_n (i.e. le couple (F_{n-1}, F_n) dont on extraira F_n. Pour cela que n soit pair ou impair on calculera T_p ave $p = n/2$ (division entière), et selon la parité de n on utilisera une expression ou l'autre de la question précédente.

   On peut donc clairement utiliser la méthode diviser pour régner, et si on programme correctement gagner beaucoup en complexité.

   On peut dès lors proposer le code suivant :

```
let fibo_dpr n =
  if n <= 1 then n else
    let rec fibo_dpr_aux n =
      if n = 1 then (0, 1)
      else
        let (fnm1, fn) = fibo_dpr_aux (n / 2) in
        if n mod 2 = 0 then (fn * fn + fnm1 * fnm1, fn * fn + 2
    * fn * fnm1)
        else (fn * fn + 2 * fn * fnm1, 2 * fn * fn + fnm1 *
   fnm1 + 2 * fn * fnm1)
    in let t = fibo_dpr_aux n in
    snd t ;;
```

4. La complexité sera celle de la fonction récursive, l'enrobage du premier appel étant en temps constant. Notons C_n la complexité de la fonction récursive. Il est clair que l'on fait un seul appel sur un paramètre deux fois plus petit et qu'en plus on fait des opérations en temps constant. On a donc une relation de récurrence du type $C_n = C_{n/2} + K$ où K est une constante. On en déduit $C_n = m \times K$ où m est le nombre d'appels, qui correspond au nombre de fois que l'on peut diviser n par 2 avant d'arriver au cas de base $n = 1$, soit $m = \ln n$. Dès lors on a donc une complexité logarithmique, ce qui est un gain énorme de complexité par rapport à la programmation naïve...

III

Le corrigé est dans le fichier `EnveloppeConvexe.ml` .