Correction : exercice 3 TD arbres

Pour constuire `compte n`, on remarque que n'importe quel entier `i` peut se retrouver comme étiquette de la racine de l'arbre binaire de recherche, puis que pour un `i` donné, le fils gauche est un arbre binaire de recherche étiqueté avec les entiers de 1 à $i-1$, le fils droit avec les entiers de $i+1$ à n. Ces deux arbres pouvant être construit de façon indépendante, on calcule le produit de nombre de possibilités à gauche et à droite pour chacun des `i`. Par conséquent, on a :

$$\begin{cases} \texttt{compte 0} = 1 \\ \texttt{compte n} = \sum_{i=1}^{n} \texttt{compte i-1} \times \texttt{compte (n-i)} \end{cases}$$

En utilisant une référence et une boucle pour la somme, cela donne :

```
let rec compte n =
  if n=0 then 1 else
    let sum = ref 0 in
    for i = 1 to n do
      sum := !sum + compte (i-1) * compte (n-i)
    done;
    !sum;;
```

Alternativement, une fonction récursive peut remplacer la boucle :

```
let rec compte n =
  (* loop k : Somme pour [1<=i<=k] de compte(i-1) * compte(n-i) *)
  let rec loop = function
    | 0 -> 0
    | k -> compte (k-1) * compte (n-k) + loop (k-1)
  in if n=0 then 1 else loop n;;
```

Pour déterminer le nombre d'arbres binaires de recherche contenant exactement les entiers de 1 à n de hauteur h, c'est un peu plus délicat. On peut commencer par déterminer le nombre d'arbres binaires de recherche de hauteur strictement inférieure à h, en écrivant, de façon similaire à la fonction précédente :

```
(* Compte le nombre d'ABR de taille n et de hauteur
                                        strictement inférieure à n *)
let rec compte_sih n h =
  (* loop k : Somme pour [1<=i<=k] de
                          compte_sih(i-1, h-1) * compte_sih(n-i, h-1) *)
  let rec loop = function
    | 0 -> 0
    | k -> compte_sih (k-1) (h-1) * compte_sih (n-k) (h-1) + loop (k-1)
  in if n=0 then (if h>=0 then 1 else 0) else loop n;;
```

Une fois cette fonction écrite, on peut obtenir le nombre d'arbres binaires de recherche de hauteur égale à h : il faut que les deux sous-arbres aient une hauteur égale à $h-1$, ou bien que l'un des sous-arbres a une hauteur égale à $h-1$ et l'autre une hauteur strictement inférieure à $h-1$ (que l'on peut dénombrer grâce à la fonction précédente). Ces trois cas sont mutuellement exclusifs, ce qui permet de les sommer. Cela donne :

```
(* Compte le nombre d'ABR de taille n et de hauteur égale à n *)
let rec compte_h n h =
  let rec loop = function
    | 0 -> 0
    | k -> compte_h (k-1) (h-1) * compte_sih (n-k) (h-1)
           + compte_sih (k-1) (h-1) * compte_h (n-k) (h-1)
           + compte_h (k-1) (h-1) * compte_h (n-k) (h-1)
           + loop (k-1)
  in if n=0 then (if h= -1 then 1 else 0) else loop n;;
```

Alternativement, les arbres de hauteur h sont les arbres ayant une hauteur strictement inférieure à $h+1$, mais pas une hauteur strictement inférieure à h, donc on peut aussi écrire :

```
let rec compte_h n h =
  compte_sih n (h+1) - compte_sih n h;;
```

Ces fonctions ont une complexité qui explose rapidement lorsque n augmente (cela dit, le problème est limité par le fait que les valeurs retournées dépassent aussi très vite la capacité des entiers en OCaml). On peut utiliser la programmation dynamique pour résoudre le problème. Par exemple, pour `compte`, avec mémoïzation avec un tableau [1] (`mem.(i)` contient `compte i` s'il a déjà été calculé, et 0 sinon) :

```
let rec compte n =
  let mem = Array.make (n+1) 0 in  (* 0 = non calculé *)
  mem.(0) <- 1;
  let rec compte_mem n =
    begin
      if mem.(n)=0 then
        let rec loop = function
          | 0 -> 0
          | k -> compte_mem (k-1) * compte_mem (n-k) + loop (k-1)
        in mem.(n) <- loop n
    end;
    mem.(n)
  in compte_mem n;;
```

Ou bien par stratégie « bottom up » où on remplit ce même tableau `mem` directement, de gauche à droite :

```
let rec compte n =
  let mem = Array.make (n+1) 0 in
  mem.(0) <- 1;
  for k = 1 to n do
    for i = 1 to k do
      mem.(k) <- mem.(k) + mem.(i-1) * mem.(k-i)
    done
  done;
  mem.(n);;
```

Cette approche est dorénavant quadratique en n.

1. On pourrait utiliser un dictionnaire également.

Pour clore, on pourra remarquer que la fonction `compte` retourne en fait le n^{e} nombre de Catalan. On peut utiliser le fait que ce nombre est

$$\frac{1}{n+1} \times \binom{2n}{n} = \frac{(n+2) \times (n+3) \times \ldots \times (2n)}{2 \times 3 \times \ldots \times n}$$

On peut donc utiliser par exemple cette solution, cette fois linéaire en n :

```
let catalan n =
  let i = ref 2 and s = ref 1 in
  for j = n+2 to 2*n do
    s := !s * j;
    while (!s mod !i = 0 && !i <= n) do s := !s / !i; incr i done;
  done;
  !s;;
```

Signalons que sur une architecture dont les entiers positifs sont sur 62 bits, `cat` donne un résultat correct jusque $n = 30$ (au-delà il y a débordement), mais la fonction `compte` donne le bon résultat jusque $n = 35$ (pour $n > 35$, le résultat n'est de toute façon pas représentable). On pourrait améliorer encore cette dernière fonction en déterminant via un cribble les facteurs premiers des entiers de 2 à $2n$ pour travailler avec leurs multiplicités et éviter le débordement (c'est surtout intéressants pour de grands n lorsque l'on s'intéresse simplement au reste de la division du n^{e} nombre de Catalan par un entier m connu).