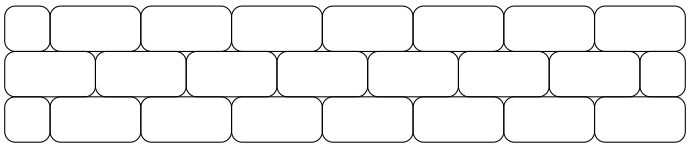
Colle 9

Compter les briques

MPII : Ocaml

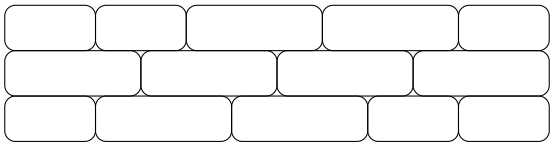
Léon le maçon est un artiste et il est malheureux avec les briques classiques avec lesquelles il répète toujours le même motif.



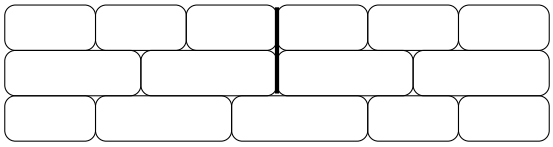
Sa vie change quand il peut utiliser des briques de tailles 2 ou 3 : nous allons voir quelles possibilités s'offrent à lui dans ce cas

I Formulation du problème

On cherche à construire un mur à partir de briques horizontales de taille 2x1 et 3x1. Un mur est correctement construit si la jointure verticale entre deux briques ne se trouve jamais immédiatement au-dessus d'une autre jointure verticale. Ainsi le mur suivant est correctement construit :



En revanche, celui-ci ne l'est pas :



L'objectif du TP est de compter le nombre de façons différentes de construire correctement un mur de largeur et de hauteur données.

II Rangées possibles

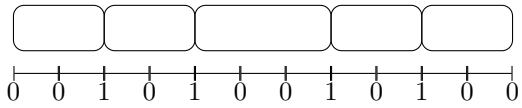
Pour dénombrer les murs on les représente comme des ensembles de rangées de briques.
On remarque qu'une rangée de longueur $n \geqslant 3$ est obtenue

- soit en ajoutant une brique de taille 2 à droite d'une rangée de taille $n-2$,
- soit en ajoutant une brique de taille 3 à droite d'une rangée de taille $n-3$,.

Exercice 1
Écrire une fonction `nb_rangees : int -> int` qui calcule le nombre de rangées possibles de largeur n donnée en paramètre.

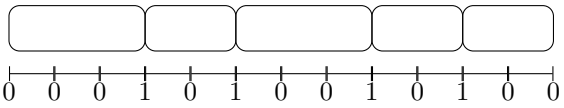
`nb_rangees 50` donne 525456.
Dans tous le TP les calculs qui appliquent les récursivité directement demandent beaucoup de calculs, par exemple `nb_rangees 75` commence à demander beaucoup de temps. Si c'est le cas reprenez votre algorithme en appliquent les méthodes de la programmation dynamique.

Une rangée de largeur n sera représentée par série de $n+1$ bits : le i-ième bit indique si, au niveau de la i-ième unité de largeur, il y a ou non une jointure entre deux briques. Les deux extrémités seront toujours des bits 0.
Ainsi la rangée de briques de largeur 11



est représentée par les 12 bits 001010010100.
En Caml ce nombre est représenté par l'entier `660 = 0b001010010100`.

- Le nombre est toujoujours divisible par 4 car les deux derniers bits sont toujours nuls.
- Les premiers 0 ne sont pas visibles ; 660 représente aussi la rangée de taille 12



même si celle-ci est représentée par 13 bits : 0001010010100.

Pour simuler l'ajout d'une brique, on décale les bits vers la gauche en utilisant l'opérateur `lsl`, `n lsl p` renvoie $n.2^p$ (avec débordement possible si on dépasse `max_int`.

À retenir - Application d'une fonction à une liste

La fonction `List.map` du module `List` applique une fonction aux éléments d'une liste et renvoie la liste des images : si `l = [a; b; c; ...; x]`, `List.map f l` renvoie la liste `l = [f a; f b; f c; ...; f x]`

```
List.map : ('a -> 'b) -> 'a list -> 'b list
```

Exercice 2
Écrire une fonction `rangees : int -> int list` qui calcule la liste des rangées possibles de largeur n sous forme d'entiers les représentant..
On prendra soin de minimiser la complexité temporelle et aussi la complexité spatiale.

II.1 Rangées compatibles

On voit que deux rangées sont superposable si et seulement si elles ont même longueur et si elles n'ont pas chacune un 1 à la même place. Ceci peut se vérifier simplement sur la représentation en base 2 en faisant le produit des bits ce qui correspond à un **et** logique en assimilant 0 et `false` ainsi que 1 et `true`.
Caml fournit un opérateur effectuant le et bit-par-bit : `land`.
Par exemple `11 land 14` renvoie 10 car les représentations en base 2 donnent
`0b00001011 land 0b00001110 -> 0b00001010`.
Ainsi deux rangées de même taille représentées par n_1 et n_2 seront compatibles si et seulement si `n1 land n2` vaut 0.

III Nombre de murs

On veut calculer le nombre de murs de largeur n et de hauteur h constructibles, c'est-à-dire tels que chaque rangée soit compatible avec la rangée du dessous.
On note R_n l'ensemble des rangées possibles de largeur n et ρ_n leur nombre.
On note $N(n,h)$ le nombre de murs constructibles de hauteur h et de largeur b.

Une idée itérative serait de calculer tous les assemblages de h rangées de largeur n et de compter ceux qui sont constructibles : la complexité promise, $(h-1).\rho_n^h$ est dissuasive.
On va donc essayer un algorithme récursif.
On note $N_1(n,h,r)$ le nombre de constructions possibles, de hauteur h et de largeur n, au-dessus d'une rangée e. On a donc $N(n,h) = \sum_{r \in R_n} N_1(n, h-1, r)$.
Si on remarque que toute rangée est compatible avec la rangée (fictive) de numéro 0, on peut aussi écrire $N(n,h) = N_1(n,h,0)$.
La récursivité porte sur h : pour construire un mur de hauteur h au dessus de r, on doit construire un mur de hauteur $h-1$ au-dessus d'une rangée r' compatible avec r :

$$N_1(n,h,r) = \sum_{\substack{r' \in R_n \\ r \,\texttt{land}\, r' = 0}} N_1(n, h-1, r')$$

Exercice 3
Écrire une fonction `nb_murs larg haut` qui renvoie le nombre de murs de largeur `larg` et de hauteur `haut`.

`nb_murs 15 15` renvoie 27 483 324 ; le temps de calcul se compte en dizaines de secondes.
`nb_murs 16 16` renvoie 327 019 964 ; le temps de calcul se compte en minutes.

IV Tables de hachage

Le programme ci-dessus est correct mais sa complexité est mauvaise. Ceci est dû au fait qu'à cause des appels récursifs, le calcul de $N_1(n,h,r)$ est effectué de très nombreuses fois pour les mêmes valeurs. Pour régler le problème, il suffit de mémoriser la valeur une fois qu'on l'a calculée. Les fois suivantes, on ira directement chercher le résultat sans le recalculer.
On a donc besoin d'une structure de données de type clé-valeur (un dictionnaire) dont les clés sont des couples `(h,r)` (la largeur est constante) et la valeur est le nombre de murs constructibles, `nb`.
Nous allons l'implémenter sous forme d'une table de hachage.
Nos éléments sont donc des triplets d'entiers `(h, r, nb)`.

- On définit une fonction de hachage `hash` qui associe un entier à tout couple `(h, r)` un entier. On essaiera de définir une fonction qui peut discriminer raisonnablement les couples.

- On définit un tableau de longueur `taille` et on place le triplet `(h, r, nb)` à la position `i = (hash h r)mod taille`.
- Comme il est pratiquement impossible de s'assurer que deux éléments seront toujours associés à des indices distincts, les éléments du tableau sont des listes. On espère que ces listes ne seront pas trop longues. Pour placer un nouveau triplet à la position `i` on l'adjoint à la liste.

IV.1 Fonctions

Pour chercher un élément `(h, r, n)` dans une table de hachage `table_h` on calcule la valeur hachée `i = (hash h r)mod taille` puis on cherche l'élément dans la liste `table_h.(i)`.
On choisit en général `taille` premier : ici on pourra prendre 5003.
`taille` est donné comme une variable globale.

Exercice 4
Écrire les fonctions

1. `mem h r liste` qui renvoie `true` ou `false` selon qu'il existe ou non un élément de la forme `(h, r, nb)` dans la liste.
2. `assoc h r liste` qui renvoie la valeur `nb` telle que `(h, r, nb)` est dans la liste **si** un tel élément existe. La fonction renverra une erreur sinon.

Exercice 5
Écrire les fonctions

1. `hash h r` qui donne un entier positif *raisonnable* à partir de h et r
2. `cree_table_h ()` qui crée une table de taille `taille` (une variable globale) dont les éléments sont des listes vides.
3. `existe h r table_h` qui teste s'il existe un triplet dans la table dont les premières composantes sont h et r
4. `valeur h r table_h` qui renvoie la valeur de n de l'élément `(h,r,n)` dans la table.
5. `ajoute h r n table_h` qui ajoute un triplet `e = (h,r,n)` à la table `table_h` s'il n'y avait pas d'élément de la forme `(h, r, p)` dans la table.

IV.2 Dénombrement

Exercice 6
Modifier la fonction `nb_murs n h` pour ne pas recalculer les mêmes valeurs plusieurs fois.

Admirer l'accélération !

Solutions

Solution de l'exercice 1

```
let rec nb_rangees n =
   match n with
   |1 -> 0
   |2 -> 1
   |3 -> 1
   |_ -> nb_rangees (n-2) + nb_rangees (n-3);;
```

Si on mémorise les derniers calculs, on va plus vite.

```
let nb_r n =
   let rec aux k a b c =
      if k = n
      then c
      else aux (k+1) b c (a+b) in
   aux 3 0 1 1;;
```

Solution de l'exercice 2

On commence par les écritures des fonctions d'ajout

```
let decaler k n = (n+1) lsl k;;

let plus2 = decaler 2;;

let plus3 = decaler 3;;
```

```
let rangees n =
   let rec aux k a b c =
      if k = n
      then c
      else aux (k+1) b c ((List.map plus3 a) @ (List.map plus2 b))
          in
   aux 3 [] [0] [0];;
```

Solution de l'exercice 3

```
let nb_murs larg haut =
   let rg = rangees larg in
   let rec aux h base =
    let rec aux1 liste =
      match liste with
      |[] -> 0
      |t::q -> if (base land t) = 0
               then (aux (h-1) t) + (aux1 q)
               else aux1 q in
    if h = 0 then 1 else aux1 rg
  in aux haut 0;;
```

Solution de l'exercice 4
Ce sont des fonctions classiques, ici elles sont appliquées à un couple.
Les deux fonctions ont la même structure.

```
let rec mem h r liste =
  match liste with
  |[] -> false
  |(a, b, c)::q -> if a = h && b = r
                   then true
                   else mem h r q;;
```

```
let rec assoc h r liste =
  match liste with
  |[] -> failwith "L'element n'est pas dans la table"
  |(a, b, c)::q -> if a = h && b = r
                   then c
                   else assoc h r q;;
```

Solution de l'exercice 5
On a vu que les nombres représentant les rangées étaient divisibles par 4 : il vaut mieux les diviser par 4.

```
let taille = 5003;;
```

```
let hash h r =
   r/4 + h*178543;;
```

```
let cree_table_h () =
   Array.make taille [];;
```

```
let existe h r table_h =
   let i = (hash h r) mod taille
   in mem h r table_h.(i);;
```

```
let valeur h r table_h =
   let i = (hash h r) mod taille
   in assoc h r table_h.(i);;
```

```
let ajoute h r n table_h =
   if not(existe h r table_h)
   then let i = (hash h r) mod taille
        in table_h.(i) <- (h,r,n)::table_h.(i);;
```

Solution de l'exercice 6

```
let nb_murs larg haut =
  let rg = rangees larg in
  let ht = cree_table_h () in
  let rec aux h base =
    if existe h base ht
    then valeur h base ht
    else begin
           let rec aux1 liste =
             match liste with
             |[] -> 0
             |t::q -> if (base land t) = 0
                      then (aux (h-1) t) + (aux1 q)
                      else aux1 q in
           let k = if h = 0 then 1 else aux1 rg in
           ajoute h base k ht;
           k end
  in aux haut 0;;
```