Chapitre 8

Correction des exercices

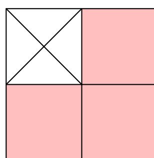
Exercice 1 Nous allons écrire une fonction calculant une suite décroissante de segments $[\![i_p, j_p]\!]$ tel que pour tout p, $i_p^2 \leqslant n < j_p^2$. Lorsque $j = i+1$ nous auront trouvé l'entier recherché.

```
let isqrt n =
  let rec cherche i j =
    let k = (i + j) / 2 in
    match k * k with
      | _ when j = i + 1 -> i
      | x when x > n     -> cherche i k
      | x when x = n     -> k
      | _                -> cherche k j
  in cherche 0 (n+1) ;;
```

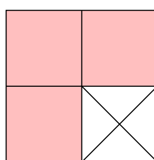
Dans le pire des cas, 3 comparaisons entre entiers et un produit sont effectués à chaque étape, ce qui conduit à une relation de récurrence de la forme $c_n = c_{\lceil n/2 \rceil} + \Theta(1)$, soit : $c_n = \Theta(\log n)$.

Exercice 2

a) Si $z = b(i_0, j_0)$, il est facile de constater que pour tout $i \leqslant i_0$ et $j \leqslant j_0$ on a $b(i,j) \leqslant z$. À l'inverse, pour tout $i \geqslant i_0$ et $j \geqslant j_0$ on a $b(i,j) \geqslant z$.
Ainsi, si $v > x$ on peut chercher v dans la partie du tableau correspondant aux indices $i \geqslant \dfrac{n}{2}$ et $j \geqslant \dfrac{n}{2}$.

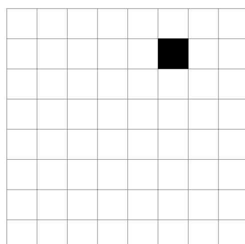


À l'inverse, si $v < y$ on peut chercher v dans la partie du tableau correspondant aux indices $i \leqslant \dfrac{n}{2}+1$ et $j \leqslant \dfrac{n}{2}+1$.

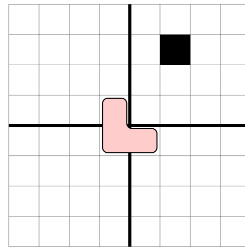


b) Sachant que $x \leqslant y$, l'une de ces deux conditions est forcément vérifiée (ou alors c'est qu'on a trouvé v). Dans le pire des cas, après ces deux tests la recherche se ramène à trois tableaux de tailles $\dfrac{n}{2} \times \dfrac{n}{2}$. Le coût dans le pire des cas vérifie donc la relation : $c_n = 3c_{n/2} + \Theta(1)$; il s'agit d'un coût en $\Theta(n^{\log 3})$.

Exercice 3 Lorsque $k = 1$, chacun des quatre motifs permet de recouvrir l'un des échiquiers 2×2 privé d'une case.
Lorsque $k > 1$, supposons que l'on sache recouvrir tout échiquier $2^{k-1} \times 2^{k-1}$ privé de n'importe laquelle de ses cases, et considérons un échiquier $2^k \times 2^k$ privé d'une case :



Cet échiquier peut être séparé en quatre échiquiers $2^{k-1} \times 2^{k-1}$, l'un d'entre eux (dans mon exemple celui situé au nord-est) étant amputé d'une case. Parmi les quatre motifs à notre disposition, un seul peut être placé de manière à recouvrir les trois autres échiquiers :



On constate alors qu'il reste à recouvrir quatre échiquiers $2^{k-1} \times 2^{k-1}$ amputés chacun d'une case ; c'est le sens de l'appel récursif.

Exercice 4

a) L'algorithme naïf consiste à tester tous les couples (i, j) vérifiant $1 \leqslant i < j \leqslant n$, ce qui donne :

```
let inversion t =
  let n = vect_length t in
  let s = ref 0 in
  for i = 0 to n-2 do
    for j = i+1 to n-1 do
      if t.(i) > t.(j) then s := !s + 1
    done
  done ;
  !s ;;
```

Le nombre de comparaisons effectuées entre éléments du tableau vaut : $c_n = \sum_{i=0}^{n-2}(n-1-i) = \frac{n(n-1)}{2} = \Theta(n^2)$, le coût de cet algorithme est bien quadratique.

b) Adopter une stratégie « diviser pour régner » consiste à séparer le tableau en deux parties sensiblement égales de longueur k et $n-k$, à calculer récursivement le nombre d'inversions présentes dans chacune des deux parties, puis à déterminer les inversions non comptabilisées : celles qui sont à cheval sur les deux parties, c'est à dire pour lesquelles $i \leqslant k < j$. D'après le théorème maître, ceci doit pouvoir être réalisé en temps linéaire pour faire mieux que l'algorithme naïf. Nous allons montrer que ceci est possible lorsque les parties gauche et droite sont *triées*. Ainsi, nous allons supposer que les appels récursifs retournent non seulement le nombre d'inversions mais aussi la liste triée (par ordre croissant) des éléments.

Nous aurons besoin d'une fonction prenant en argument deux listes triées (y_i) et (z_j) et dénombrant le nombre de couples (y_i, z_j) vérifiant $y_i > z_j$:

```
let acheval l1 l2 =
  let rec aux acc n = fun
    | [] _ -> acc
    | _ [] -> acc
    | y z when hd y > hd z -> aux (n + acc) n y (tl z)
    | y z -> aux acc (n-1) (tl y) z
  in aux 0 (list_length l1) l1 l2 ;;
```

Il est clair que cette fonction a un coût linéaire vis-à-vis de $|y|+|z|$ puisque chaque appel récursif diminue la longueur d'une des deux listes d'une unité. La validité de cette fonction résulte de la constatation suivante : si $y = (y_1 \leqslant y_2 \leqslant \cdots \leqslant y_n)$ et $z = (z_1 \leqslant z_2 \cdots \leqslant z_p)$, alors :

– si $y_1 > z_1$ les n couples (y_i, z_1) vérifient $y_i > z_1$;

– si $y_1 \leqslant z_1$; aucun couple (y_1, z_j) ne vérifie $y_1 < z_j$.

La fonction générale se définit alors ainsi (la fonction **fusion** est définie dans le cours ; elle a pour objet de fusionner deux listes triées en coût linéaire) :

```
let inversionbis t =
  let rec aux = function
    | (i, j) when i = j -> 0, [t.(i)]
    | (i, j) -> let k = (j + i) / 2 in
                let (c1, y) = aux (i, k) and (c2, z) = aux (k+1, j) in
                (c1 + c2 + acheval y z), (fusion y z)
  in match (vect_length t) with
    | 0 | 1 -> 0
    | n -> fst (aux (0, n-1)) ;;
```

Le coût de cette fonction vérifie la relation : $c_n = c_{\lfloor n/2 \rfloor} + c_{\lceil n/2 \rceil} + \Theta(n)$ donc $c_n = \Theta(n \log n)$.

Exercice 5

a) Sachant qu'une matrice de Tœplitz est entièrement définie par sa première ligne et sa première colonne, l'addition de deux matrices ne nécessite que $2n-1$ additions scalaires, donc un coût linéaire.

b) Découpons une matrice de Tœplitz en 4 blocs carrés et un vecteur colonne en deux parties égales. Alors :

$$\begin{pmatrix} A & B \\ C & A \end{pmatrix} \begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} AX + BY \\ CX + AY \end{pmatrix}$$

mais on se doute bien que si on se contente d'utiliser cette formule, le gain sera nul par rapport au produit classique.

Posons donc $U = (A+C)X$, $V = A(X-Y)$, et $W = (A+B)Y$. Alors $\begin{pmatrix} A & B \\ C & A \end{pmatrix} \begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} W+V \\ U-V \end{pmatrix}$.

Les différentes additions à effectuer ont un coût linéaire (puisqu'il s'agit soit d'additions entre vecteurs, soit d'additions entre matrices de Tœplitz), donc le coût total c_n vérifie la relation : $c_n = 3c_{n/2} + \Theta(n)$, ce qui conduit à $c_n = \Theta(n^{\log 3})$.

Exercice 6 Notons c_n le nombre de multiplications effectuées par cet algorithme, et utilisons la décomposition en base 3 de $n = [b_p, \ldots, b_0]_3$.

Nous avons $c_1 = 0$, $c_2 = 1$, et au-delà : $c_n = \begin{cases} c_k + 2 & \text{si } n = 3k\,; \\ c_k + 3 & \text{si } n = 3k+1\,; \\ c_k + 3 & \text{si } n = 3k+2. \end{cases}$

(en effet, lorsque $n = 3k+2$ le calcul de x^2 **et** de x^3 ne nécessite que deux multiplications).
Sachant que $k = [b_p, \ldots, b_1]_3$, on en déduit l'encadrement : $2p \leqslant c_n \leqslant 3p+1$, soit : $2\lfloor \log_3 n \rfloor \leqslant c_n \leqslant 3\lfloor \log_3 n \rfloor + 1$.

Comparons le coût dans le pire des cas avec celui de l'algorithme binaire. Nous avons ici $c_{\max} \sim 3\log_3 n$, contre $2\log_2 n$ pour l'algorithme binaire. Sachant que $\frac{3}{\ln 3} < \frac{2}{\ln 2}$, l'algorithme ternaire est asymptotiquement plus intéressant.
En revanche, on peut montrer que le coût en moyenne de cet algorithme vérifie $c_{\text{moy}} \sim \frac{8}{3}\log_3 n$, ce qui est moins bon que l'algorithme d'exponentiation rapide (qui lui a une complexité moyenne équivalente à $\frac{3}{2}\log_2 n$) car $\frac{3}{2\ln 2} < \frac{8}{3\ln 3}$.