Chapitre 8

option informatique

Diviser pour régner

1. Introduction

1.1 Algorithme d'exponentiation rapide

Considérons un entier n et un élément x d'un monoïde multiplicatif (X,∗) (x peut donc être un nombre, une matrice, un polynôme…), et intéressons-nous au calcul de x^n, en choisissant pour mesure du coût le nombre de multiplications effectuées. Un algorithme simple vient immédiatement à l'esprit :

```
let rec puissance x = function
  | 0 -> 1
  | 1 -> x
  | n -> x * puissance x (n-1) ;;
```

(par souci de lisibilité, tous nos algorithmes seront de type *int -> int* ; pour adapter ceux-ci à un autre type il suffira de modifier l'élément neutre `1` et le produit `*`).
À l'évidence, le nombre de multiplications effectuées est égal à $n-1$; il s'agit donc d'un algorithme de coût linéaire.
Il est cependant très facile de faire mieux, en utilisant l'algorithme suivant, connu sous le nom d'algorithme d'*exponentiation rapide* :

```
let rec puissance x = function
  | 0                  -> 1
  | 1                  -> x
  | n when n mod 2 = 0 -> puissance (x * x) (n / 2)
  | n                  -> x * puissance (x * x) (n / 2) ;;
```

Il s'agit d'une fonction inductive dont la terminaison est justifiée par l'inégalité : $\forall n \geqslant 2, \left\lfloor \frac{n}{2} \right\rfloor < n$ et la validité par les égalités : $\begin{cases} x^{2k} = (x^2)^k \\ x^{2k+1} = x(x^2)^k \end{cases}$
Si on note c_n le nombre de multiplications effectuées, on dispose des relations : $c_0 = c_1 = 0$, $c_{2k} = c_k + 1$ et $c_{2k+1} = c_k + 2$, soit encore : $c_n = c_{\lfloor n/2 \rfloor} + 1 + (n \bmod 2)$.

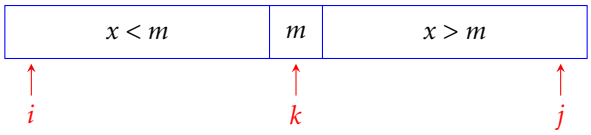
Considérons la décomposition de n en base 2 : $n = [b_p, b_{p-1}, \ldots, b_0]_2 = \sum_{k=0}^{p} b_k 2^k$, avec $b_p = 1$.

Nous avons $\left\lfloor \frac{n}{2} \right\rfloor = [b_p, b_{p-1}, \ldots, b_1]_2$ et $n \bmod 2 = b_0$, et il est alors facile d'obtenir : $c_n = p + \sum_{k=0}^{p-1} b_k$.
Le coût dans le meilleur des cas intervient lorsque n est une puissance de 2 : si $n = 2^p$, alors $c_n = p = \log n$.
Le coût dans le pire des cas intervient lorsque $n = 2^{p+1} - 1$ (l'écriture binaire de n ne comporte que des 1) : nous avons alors $c_n = 2p = 2\lfloor \log n \rfloor$.
Et il n'est guère difficile d'établir qu'en moyenne, $c_n = \frac{3}{2}\lfloor \log n \rfloor$.
Il s'agit donc d'un algorithme de coût logarithmique.

1.2 Diviser pour régner

L'algorithme d'exponentiation rapide que nous venons d'étudier est un exemple d'utilisation d'un paradigme de programmation connu sous le nom de *diviser pour régner* (ou *divide and conquer* en anglais) : il consiste à ramener la résolution d'un problème dépendant d'un entier n à la résolution de un ou plusieurs sous-problèmes identiques portant sur des entiers $n' \approx \alpha n$ avec $\alpha < 1$ (le plus souvent, on aura $\alpha = 1/2$). Par exemple, l'algorithme d'exponentiation rapide ramène le calcul de x^n au calcul de $y^{\lfloor n/2 \rfloor}$ avec $y = x^2$.

Un autre exemple classique d'utilisation de ce paradigme est l'algorithme de recherche dichotomique : étant donné un tableau trié par ordre croissant d'entiers, comment déterminer si un élément appartient ou pas à ce tableau ?
Le principe est bien connu : on compare l'élément recherché à l'élément médian, et le cas échéant on répète la recherche dans la partie gauche ou droite du tableau.




```
let recherche_dicho x t =
  let rec aux i j =
        if j < i then false
          else match (i + j) / 2 with
            | k when x = t.(k) -> true
            | k when x < t.(k) -> aux i (k-1)
            | k                -> aux (k+1) j
  in f 0 (vect_length t - 1) ;;
```

Notons c_n le nombre de comparaisons nécessaires entre x et un élément du tableau de taille n dans le pire des cas (correspondant au cas où x est supérieur à tous les éléments du tableau). Nous avons $c_0 = 0$ et $c_n = 2 + c_{\lfloor n/2 \rfloor}$ pour $n \geqslant 1$. Si on introduit de nouveau l'écriture binaire de n : $n = [b_p, b_{p-1}, \cdots, b_0]_2$, nous avons $\left\lfloor \frac{n}{2} \right\rfloor = [b_p, b_{p-1}, \cdots, b_1]_2$, et il est alors facile d'en déduire que $c_n = 2p + 2 = 2\lfloor \log n \rfloor + 2$.

1.3 Étude générale du coût

En général, les algorithmes suivant le paradigme *diviser pour régner* partagent un problème de taille n en deux sous-problèmes de tailles respectives $\left\lfloor \frac{n}{2} \right\rfloor$ et $\left\lceil \frac{n}{2} \right\rceil$ et conduisent à une relation de récurrence de la forme :

$$\boxed{c_n = a c_{\lfloor n/2 \rfloor} + b c_{\lceil n/2 \rceil} + d_n} \qquad \text{avec } a + b \geqslant 1,$$

d_n représentant le coût du partage et de la recomposition du problème et c_n le coût total.

Nous allons faire une étude générale de ces relations de récurrence en commençant par traiter le cas où n est une puissance de 2 : posons $n = 2^p$ et $u_p = c_{2^p}$. La relation de récurrence devient : $u_p = (a+b)u_{p-1} + d_{2^p}$, soit :

$$\frac{u_p}{(a+b)^p} = \frac{u_{p-1}}{(a+b)^{p-1}} + \frac{d_{2^p}}{(a+b)^p}.$$

Par télescopage on obtient : $u_p = (a+b)^p \left(u_0 + \sum_{j=1}^{p} \frac{d_{2^j}}{(a+b)^j} \right)$.

Pour poursuivre ce calcul, il est nécessaire de préciser la valeur de d_n. Nous allons supposer que le coût de la décomposition et de la recomposition est polynomial (au sens large), ce qui permet de poser $d_n = \lambda n^k$. Nous avons alors :

$$u_p = (a+b)^p \left(u_0 + \lambda \sum_{j=1}^{p} \left(\frac{2^k}{a+b} \right)^j \right) = \begin{cases} \alpha 2^{kp} + \beta (a+b)^p & \text{si } a+b \neq 2^k \\ (u_0 + \lambda p)(a+b)^p & \text{si } a+b = 2^k \end{cases}$$

en ayant posé $\alpha = \lambda \left(\frac{2^k}{2^k - a - b} \right)$ et $\beta = u_0 - \lambda \left(\frac{2^k}{2^k - a - b} \right)$.
Trois cas se présentent alors :

- si $a + b < 2^k$, $u_p \sim \alpha 2^{kp}$;
- si $a + b = 2^k$, $u_p \sim \lambda p (a+b)^p = \lambda p 2^{kp}$;
- si $a + b > 2^k$, $u_p \sim \beta (a+b)^p$.

Pour traiter le cas général d'un entier n quelconque, nous allons maintenant prouver le résultat suivant :

Lemme. — *Lorsque la suite $(d_n)_{n\in\mathbb{N}}$ est croissante, la suite $(c_n)_{n\in\mathbb{N}}$ l'est aussi.*

Preuve. Montrons par récurrence sur $n \in \mathbb{N}^*$ que $c_n \leqslant c_{n+1}$.

- Lorsque $n = 1$, on a : $c_2 = (a+b)c_1 + d_1 \geqslant c_1$ car $a + b \geqslant 1$ et $d_1 \geqslant 0$.
- Si $n > 1$, on suppose le résultat acquis jusqu'au rang $n-1$. On a alors :

$$c_{n+1} - c_n = a(c_{\lceil (n+1)/2 \rceil} - c_{\lceil n/2 \rceil}) + b(c_{\lfloor (n+1)/2 \rfloor} - c_{\lfloor n/2 \rfloor}) + d_{n+1} - d_n \geqslant 0$$

ce qui achève la démonstration.

□

Lorsque cette condition est satisfaite, on peut encadrer n par deux puissances consécutives de 2 : $2^p \leqslant n < 2^{p+1}$, avec $p = \lfloor \log n \rfloor$, ce qui conduit à $u_p \leqslant c_n \leqslant u_{p+1}$. Lorsque $d_n = \lambda n^k$, les résultats précédents permettent d'établir la règle suivante (connue sous le nom de *théorème maître*) :

Théorème. — *Lorsque $a + b > 1$, la suite $(d_n)_{n\in\mathbb{N}}$ croissante et $d_n = \Theta(n^k)$, on a :*

$$\boxed{\begin{array}{l} \text{si } \log(a+b) < k,\ c_n = \Theta(n^k)\,; \\ \text{si } \log(a+b) = k,\ c_n = \Theta(n^k \log n)\,; \\ \text{si } \log(a+b) > k,\ c_n = \Theta(n^{\log(a+b)}). \end{array}}$$

Nous utiliserons désormais ce résultat pour évaluer rapidement le coût d'une méthode suivant le principe *diviser pour régner*.

2. Exemples d'algorithmes « diviser pour régner »

2.1 Tri fusion

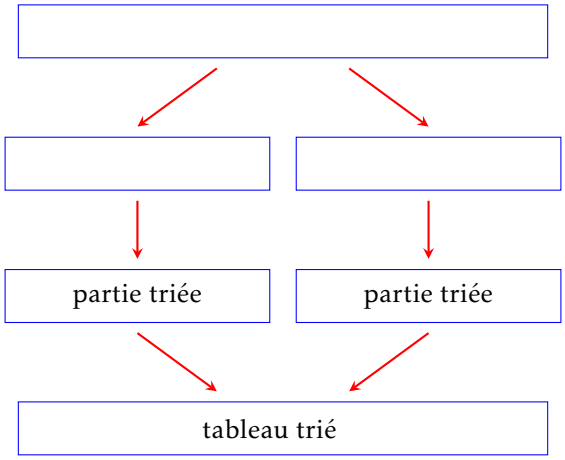
On appelle *algorithme de tri* tout algorithme permettant de trier les éléments d'un tableau ou d'une liste selon un ordre déterminé. On peut distinguer parmi ceux-ci ceux qui procèdent par comparaison entre paires d'éléments ; dans ce cas, on adopte en général comme élément de mesure de la complexité le nombre de comparaisons effectuées en fonction de la taille n du tableau (mais ce n'est pas forcément le seul paramètre à prendre en compte). Les algorithmes naïfs (tri par sélection ou par insertion par exemple) ont un coût dans le pire des cas en $O(n^2)$. Nous allons montrer qu'il est possible de faire mieux.

Appelé *merge sort* en anglais, le tri fusion adopte une approche « diviser pour régner » : on partage le tableau en deux parties de tailles respectives $\left\lfloor \frac{n}{2} \right\rfloor$ et $\left\lceil \frac{n}{2} \right\rceil$ que l'on trie par un appel récursif, puis on fusionne les deux parties triées.

– scission du tableau en deux parties :

– tri des deux moitiés du tableau :

– fusion des parties triées :



Il est raisonnable de penser que la scission et la fusion ont un coût linéaire (ce sera vérifié plus loin), et que par voie de conséquence la relation de récurrence satisfaite par le nombre c_n de comparaisons est de la forme : $c_n = c_{\lfloor n/2 \rfloor} + c_{\lceil n/2 \rceil} + \Theta(n)$. Le théorème maître montre que dans ces conditions $c_n = \Theta(n \log n)$.

Cependant, une difficulté se présente lorsqu'on cherche à implémenter cet algorithme avec des vecteurs : il est difficile de fusionner deux demi-tableaux sur place, c'est à dire en s'autorisant uniquement la permutation de deux éléments (ce n'est pas impossible, mais ou bien lent, ou bien compliqué à mettre en œuvre). Mettre en œuvre cet algorithme sur une liste est mieux adapté.

On choisit de scinder les éléments de la liste suivant la parité de leur indice :

```
let rec scission = function
  | []      -> [], []
  | [a]     -> [a], []
  | a::b::q -> let (l1, l2) = scission q in a::l1, b::l2 ;;
```

La fusion de deux listes triées ne pose aucun problème :

```
let rec fusion = fun
  | [] l2                       -> l2
  | l1 []                       -> l1
  | (t1::q1) l2 when t1 < hd l2 -> t1::(fusion q1 l2)
  | l1 (t2::q2)                 -> t2::(fusion l1 q2) ;;
```

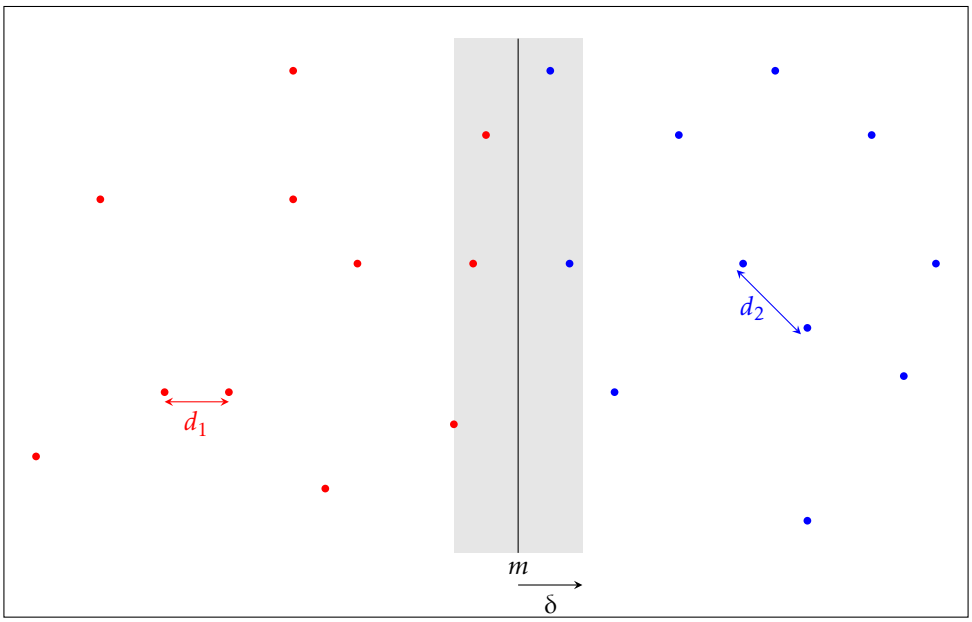
Il reste à rédiger l'algorithme de tri fusion proprement dit :

```
let rec merge_sort = function
  | []  -> []
  | [a] -> [a]
  | l   -> let (l1, l2) = scission l in
           fusion (merge_sort l1) (merge_sort l2) ;;
```

2.2 Distance minimale

Intéressons nous maintenant au problème de la recherche des deux points les plus proches dans un nuage $p_1 = (x_1, y_1), p_2 = (x_2, y_2), \dots, p_n = (x_n, y_n)$ de points distincts du plan. L'algorithme naïf consiste à calculer la distance entre toutes les paires possibles et d'en extraire le minimum. Sachant qu'il y en a $\binom{n}{2}$, ceci conduit à un algorithme de coût quadratique. Peut-on faire mieux en adoptant une stratégie diviser pour régner ?

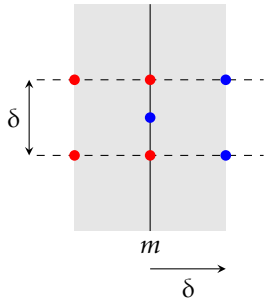
Avant toute chose, nous allons avoir besoin d'ordonner ces points par abscisse croissante, mais aussi par ordonnée croissante. Nous allons donc calculer deux tableaux $\mathcal{P}$ et $\mathcal{P}'$, le premier contenant les points classés par ordre croissant des abscisses, et le second par ordre croissant des ordonnées. D'après la section précédente, nous savons que ceci peut être exécuté en $\Theta(n \log n)$.
Nous allons désormais supposer $\mathcal{P} = (p_1, p_2, \dots, p_n)$ avec $x_1 \leqslant x_2 \leqslant \cdots \leqslant x_n$.



Séparons alors les points de $\mathcal{P}$ en deux parties sensiblement égales $\mathcal{P}_1 = \{p_1, p_2, \ldots, p_k\}$ et $\mathcal{P}_2 = \{p_{k+1}, \ldots, p_{n-1}, p_n\}$, avec $k = \lfloor n/2 \rfloor$, et notons d_1 et d_2 les distances minimales entre respectivement deux points de $\mathcal{P}_1$ et deux points de $\mathcal{P}_2$.
Trois cas sont possibles pour la distance minimale d entre deux points de $\mathcal{P}$:

1. elle est atteinte entre deux points de $\mathcal{P}_1$, auquel cas $d = d_1$;
2. elle est atteinte entre deux points de $\mathcal{P}_2$, auquel cas $d = d_2$;
3. elle est atteinte entre un point de $\mathcal{P}_1$ et un point de $\mathcal{P}_2$.

Le calcul de $\delta = \min(d_1, d_2)$ permet d'éliminer l'un ou l'autre des deux premiers cas. Et dans le troisième cas, les deux points que nous recherchons se trouvent dans la bande verticale délimitée par les abscisses $m - \delta$ et $m + \delta$ avec $m = \dfrac{x_k + x_{k+1}}{2}$.
Nous pouvons en coût linéaire déterminer les points $\mathcal{P}'' = (p'_i, \ldots, p'_j)$ de cette bande verticale en les ordonnant par ordonnée croissante à l'aide du tableau $\mathcal{P}'$ (il suffit de parcourir une fois $\mathcal{P}'$ en ne gardant que les points dont les abscisses sont comprises entre $m - \delta$ et $m + \delta$).
Pour conclure, il reste à observer (c'est le point crucial) que dans chaque tranche de hauteur δ de cette bande ne peuvent se trouver qu'*au plus* sept points :



En effet, s'il en existait huit, il y en aurait deux au moins appartenant tout deux à $\mathcal{P}_1$ ou tous deux à $\mathcal{P}_2$ et à une distance strictement inférieure à δ.
Cette remarque permet de restreindre la recherche de la distance minimale entre deux points de $\mathcal{P}''$ en se restreignant aux points distants au plus de sept cases de ce tableau, ce qui conduit à un coût linéaire.

En définitive, le coût de cet algorithme (sans compter le pré-traitement des tableaux $\mathcal{P}$ et $\mathcal{P}'$) vérifie une relation de récurrence de la forme : $c_n = c_{\lfloor n/2 \rfloor} + c_{\lceil n/2 \rceil} + \Theta(n)$, ce qui prouve que $c_n = \Theta(n \log n)$.

3. Multiplication rapide

Dans cette section, nous allons appliquer la paradigme « diviser pour régner » au calcul d'un produit, d'abord de deux polynômes, puis de deux entiers, et enfin de deux matrices.

3.1 Multiplication de deux polynômes

Nous choisissons de représenter le polynôme $P = \sum_{k=0}^{n} a_k X^k \in \mathbb{Z}[X]$ par le tableau (de taille $n+1$) :

a_0	a_1	a_2	.....	a_{n-1}	a_n

(a_k étant stocké dans la case d'indice k), de sorte de pouvoir accéder à chaque coefficient avec un coût constant.
La formule $\sum_{k=0}^{n} a_k X^k + \sum_{k=0}^{n} b_k X^k = \sum_{k=0}^{n} (a_k + b_k) X^k$ montre que l'addition de deux polynômes de degré n nécessite $n+1$ additions (donc un coût linéaire).

```
let som p q =
  let n = vect_length p - 1 in
  let r = make_vect (n + 1) 0 in
  for i = 0 to n do r.(i) <- p.(i) + q.(i) done ;
  r ;;
```

Quant au produit, la formule $\left(\sum_{i=0}^{n} a_i X^i\right) \times \left(\sum_{j=0}^{n} b_j X^j\right) = \sum_{i=0}^{n}\sum_{j=0}^{n} a_i b_j X^{i+j}$ montre qu'il peut être réalisé à l'aide de n^2 additions et $(n+1)^2$ multiplications (donc un coût quadratique).

```
let prod p q =
  let n = vect_length p - 1 in
  let r = make_vect (2 * n + 1) 0 in
  for i = 0 to n do
    for j = 0 to n do
      r.(i+j) <- r.(i+j) + p.(i) * q.(j)
    done ;
  done ;
  r ;;
```

On convient généralement qu'une addition est beaucoup rapide à effectuer qu'un produit ; nous allons donc chercher à réduire le nombre de multiplications, quitte à faire plus d'additions. (Il faut cependant savoir que ceci était surtout vrai dans les temps anciens de l'informatique ; c'est beaucoup moins vrai aujourd'hui.)

Pour appliquer le paradigme « diviser pour régner », nous allons poser $m = \left\lceil \frac{n}{2} \right\rceil$ puis $P = X^m P_1 + P_2$ et $Q = X^m Q_1 + Q_2$, le degré de chacun des quatre polynômes P_1 , P_2, Q_1 et Q_2 étant inférieur ou égal à $\left\lfloor \frac{n}{2} \right\rfloor$.

Ainsi, $PQ = X^{2m} P_1 Q_1 + X^m (P_1 Q_2 + P_2 Q_1) + P_2 Q_2$.

Puisqu'on accède aux coefficients de chaque polynôme avec un coût constant, il n'y a pas de coût de décomposition. 4 appels récursifs sont nécessaires, et la recomposition nécessite 3 additions de coût linéaire. Le nombre de multiplications vérifie donc la relation :

$$c_n = 4c_{\lfloor n/2 \rfloor} + \Theta(n).$$

Le théorème maître nous permet dès lors d'affirmer que $c_n = \Theta(n^2)$, ce qui n'est pas meilleur que l'algorithme naïf. Nous n'avons donc pour l'instant rien gagné à mettre en œuvre le paradigme « diviser pour régner ».

En revanche, si on observe que : $P_1 Q_2 + P_2 Q_1 = (P_1 + P_2)(Q_1 + Q_2) - P_1 Q_1 - P_2 Q_2$, on a alors :

$$\boxed{PQ = X^{2m} R_1 + X^m (R_2 - R_1 - R_3) + R_3} \qquad \text{avec} \qquad R_1 = P_1 Q_1, \quad R_2 = (P_1 + P_2)(Q_1 + Q_2), \quad R_3 = P_2 Q_2,$$

et il n'est plus nécessaire que d'utiliser 3 appels récursifs. La relation de récurrence prend cette fois la forme :

$$c_n = 3c_{\lfloor n/2 \rfloor} + \Theta(n)$$

qui conduit à $c_n = \Theta(n^{\log 3})$ (avec $\log 3 \approx 1,585$).

Cette méthode porte le nom d'algorithme de Karatsuba ; elle a été publiée pour la première fois en 1962.

• Mise en œuvre pratique.

Pour faciliter la mise en œuvre, on suppose que les polynômes sont stockés dans des tableaux de taille 2^k (quitte à rajouter des 0), donc de degré inférieurs ou égaux à $2^k - 1$. La fonction qui suit devra donc être précédée d'une fonction « normalisant » les polynômes, et suivie d'une fonction simplifiant le résultat (c'est à dire enlevant les zéros redondants).

```
let rec prod p q =
  match (vect_length p) with
    | 1 -> [| p.(0)*q.(0) ; 0 |]
    | n -> let m = n / 2 in
           let p1 = sub_vect p m m and p2 = sub_vect p 0 m in
           let q1 = sub_vect q m m and q2 = sub_vect q 0 m in
           let r1 = prod p1 q1 and r2 = prod (som p1 p2) (som q1 q2)
             and r3 = prod p2 q2 in
           let s = make_vect (2*n) 0 in
           for i = 0 to n - 1 do
             s.(n+i) <- s.(n+i) + r1.(i) ;
             s.(m+i) <- s.(m+i) + r2.(i) - r1.(i) - r3.(i) ;
             s.(i) <- s.(i) + r3.(i)
             done ;
           s ;;
```

3.2 Multiplication de deux entiers

Les entiers que nous allons considérer sont des entiers codés en base 2 sur n bits : $x = \sum_{k=0}^{n-1} x_k 2^k$, que nous représenterons par des vecteurs à n composantes. La similitude qui existe avec les polynômes va nous permettre d'adapter les algorithmes précédents aux entiers, la seule différence consistant dans la gestion de la propagation de la retenue.
Si $x = \sum_{i=0}^{n-1} x_i 2^i$ et $y = \sum_{i=0}^{n-1} y_i 2^i$, alors $s = x + y$ va être représenté par un vecteur de longueur $n+1$ défini par les relations :

- $r_0 = 0$;
- $\forall i \in [\![0, n-1]\!], \quad s_i = (r_i + x_i + y_i) \bmod 2$;
- $\forall i \in [\![0, n-1]\!], \quad r_{i+1} = (r_i + x_i + y_i)/2$;
- $s_n = r_n$.

où r_i représente la retenue entrante correspondant à la position i.
Le coût d'une addition est donc un $\Theta(n)$.

L'algorithme naïf du produit correspond au développement : $xy = \sum_{i=0}^{n-1} (x2^i) y_i$, qui se ramène à n additions d'entiers de longueur $2n$, puisqu'une multiplication par 2 s'obtient en représentation binaire par simple décalage. Ceci donne un coût en $\Theta(n^2)$.
L'algorithme de multiplication rapide va consister à poser $x = a2^m + b$ et $y = c2^m + d$ avec $m = \lceil n/2 \rceil$ puis à calculer le produit xy par la formule : $xy = ac2^{2m} + \Big((a+b)(c+d) - ac - bd\Big)2^m + bd$, ce qui conduit au calcul de trois produits d'entiers de longueur m. Comme précédemment, le coût va être un $\Theta\Big(n^{\log 3}\Big)$.

Remarque. Historiquement, l'algorithme de Karatsuba a été le premier algorithme de multiplication asymptotiquement plus rapide que celui que nous apprenons tous à l'école primaire. Il a depuis été battu par d'autres algorithmes, celui de Toom-Cook (qui en est un raffinement) et surtout par l'algorithme de Schönhage-Strassen, qui utilise la transformation de Fourier rapide. Évidemment, toutes ces méthodes ne deviennent intéressantes dans la pratique que pour multiplier de très grands entiers entre eux.

3.3 Multiplication de deux matrices

Intéressons-nous pour finir au produit matriciel : si M et N sont deux matrices de tailles $n \times n$, leur addition nécessite à l'évidence n^2 additions scalaires, et leur produit par l'algorithme naïf, n^3 multiplications et $n^2(n-1)$ additions.

Peut-on améliorer la complexité du produit en adoptant une méthode « diviser pour régner » ?

Supposons $n = 2m$ et posons $M = \begin{pmatrix} A_1 & B_1 \\ C_1 & D_1 \end{pmatrix}$ et $N = \begin{pmatrix} A_2 & B_2 \\ C_2 & D_2 \end{pmatrix}$, où $A_1, B_1, C_1, D_1, A_2, B_2, C_2, D_2$ sont des matrices $m \times m$. Le produit MN est calculé par :

$$MN = \begin{pmatrix} A_1A_2 + B_1C_2 & A_1B_2 + B_1D_2 \\ C_1A_2 + D_1C_2 & C_1B_2 + D_1D_2 \end{pmatrix}$$

et nécessite *a priori* 8 produits et 4 additions sur des matrices $m \times m$. Utiliser ces formules conduit donc à une relation de la forme : $c_n = 8c_{\lceil n/2 \rceil} + \Theta(n^2)$, ce qui conduit à un coût en $\Theta(n^{\log 8}) = \Theta(n^3)$. On ne gagne donc rien par rapport à l'algorithme naïf.

L'algorithme de Strassen utilise les formules suivantes (qu'on laissera le soin au lecteur de vérifier) :

$$\begin{array}{lll} M_1 = (B_1 - D_1)(C_2 + D_2) & M_5 = A_1(B_2 - D_2) & X = M_1 + M_2 - M_4 + M_6 \\ M_2 = (A_1 + D_1)(A_2 + D_2) & M_6 = D_1(C_2 - A_2) & Y = M_4 + M_5 \\ M_3 = (A_1 - C_1)(A_2 + B_2) & M_7 = (C_1 + D_1)A_2 & Z = M_6 + M_7 \\ M_4 = (A_1 + B_1)D_2 & & T = M_2 - M_3 + M_5 - M_7 \end{array}$$

et dans ce cas, $MN = \begin{pmatrix} X & Y \\ Z & T \end{pmatrix}$. On peut observer que ces formules n'utilisent plus que 7 produits, ce qui donne un coût en $\Theta(n^{\log 7})$ (avec $\log 7 \approx 2,807$), meilleur (en théorie en tout cas) que l'algorithme naïf.

4. Exercices

Exercice 1 Utiliser le principe de la recherche dichotomique pour rédiger une fonction Caml calculant la racine carrée entière d'un entier $n \in \mathbb{N}$, à savoir $\lfloor \sqrt{n} \rfloor$, avec un coût logarithmique.

Exercice 2 Dans cet exercice, on considère un tableau d'entiers bi-dimensionnel b tel que chaque ligne et chaque colonne soit rangée par ordre croissant. Voici un exemple d'un tel tableau, à 4 lignes et 5 colonnes :

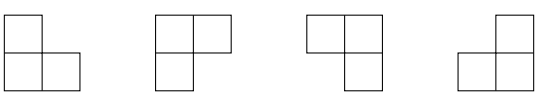
2	14	25	30	69
3	15	28	30	81
7	15	32	43	100
20	28	36	58	101

Le but de l'exercice est de rechercher efficacement un élément v dans un tel tableau.
Pour simplifier, on supposera que ce tableau comporte $n = 2^k$ lignes et colonnes.

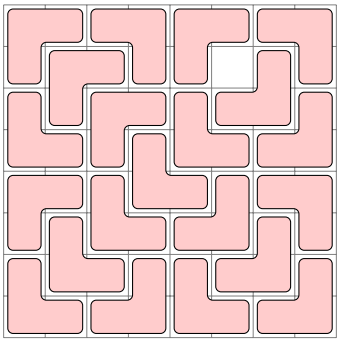
a) On distingue les deux valeurs $x = b\left(\frac{n}{2}, \frac{n}{2}\right)$ et $y = b\left(\frac{n}{2} + 1, \frac{n}{2} + 1\right)$. Montrer que si $v > x$, on peut éliminer une partie (à préciser) du tableau pour poursuivre la recherche.
Préciser ce qu'il est possible de faire lorsque $v < y$.

b) En déduire une méthode « diviser pour régner » pour résoudre ce problème, et préciser le coût dans le pire des cas, en nombre de comparaisons, de cette méthode.

Exercice 3 On considère un échiquier carré de côté $n = 2^k$, que l'on souhaite paver avec les quatre motifs ci-dessous :



Sachant que $4^k \equiv 1 \pmod 3$, ce n'est bien évidemment pas possible, mais nous allons montrer que si on isole une case de coordonnées (x, y) choisie arbitrairement, le reste de l'échiquier peut alors être recouvert :



Montrer en raisonnant par récurrence sur k qu'un tel recouvrement est toujours possible, en décrivant un algorithme « diviser pour régner » résolvant ce problème.

Exercice 4 Étant donnée une suite finie d'entiers $x = (x_1, \dots, x_n)$, on appelle *inversion* de x tout couple (i, j) tel que $i < j$ et $x_i > x_j$. Par exemple, $(2, 3, 1, 5, 4)$ possède 3 inversions : les couples $(1, 3)$, $(2, 3)$, $(4, 5)$. On s'intéresse au calcul du nombre d'inversions de x.

a) Rédiger en Caml l'algorithme naïf, et montrer que son coût est un $\Theta(n^2)$. On représentera les suites finies d'entiers par le type *int vect*.

b) Adopter une méthode « diviser pour régner » pour faire mieux (indication : adapter l'algorithme de tri fusion).

Exercice 5 Une matrice de Tœplitz est une matrice $(a_{i,j}) \in \mathcal{M}_n(\mathbb{K})$ telle que $a_{i,j} = a_{i-1,j-1}$ pour $2 \leqslant i, j \leqslant n$.

a) Trouver un moyen d'additionner deux matrices de Tœplitz en $\Theta(n)$.

b) Appliquer une méthode « diviser pour régner » pour calculer le produit d'une matrice de Tœplitz $n \times n$ par un vecteur de longueur n (en supposant pour simplifier que $n = 2^k$).
Quel est la complexité de votre algorithme ?

Exercice 6 On pourrait imaginer remplacer, dans l'algorithme d'exponentiation rapide, la base 2 par la base 3 et exploiter les relations suivantes :

$$x^n = \begin{cases} (x^3)^k & \text{si } n = 3k\,; \\ x(x^3)^k & \text{si } n = 3k+1\,; \\ x^2(x^3)^k & \text{si } n = 3k+2. \end{cases}$$

Déterminer le coût d'un tel algorithme, et le comparer au coût de l'algorithme binaire.