Chapitre 4

Diviser pour régner

1 Présentation théorique

1.1 Principe

La méthode "diviser pour régner" est un paradigme de programmation permettant généralement d'améliorer le temps de calcul des algorithmes. Le principe de base de ce paradigme consiste à ramener la résolution d'un problème dépendant d'un entier n à la résolution de plusieurs sous-problèmes identiques dépendant d'un entier $n' \sim \alpha n$, avec $\alpha \in [0, 1[$.

1.2 L'équation de complexité d'un "diviser pour régner"

Nous avons à résoudre un problème dépendant d'un entier n. Pour appliquer la méthode "diviser pour régner", nous allons :

- **partitionner** le problème en un certain nombre q de sous-problèmes de taille $n' \sim \alpha n$, avec $\alpha \in [0, 1[$;
- **résoudre** ces q sous-problèmes de taille approximative n' ;
- **fusionner** les résultats des sous-problèmes pour résoudre le problème initial.

Notons $P(n)$ le temps suffisant de partitionnement et $F(n)$ le temps suffisant de fusion, de sorte qu'on obtient une équation de complexité de la forme :

$$C(n) = P(n) + qC(n') + F(n).$$

Dans la pratique, on travaille avec $\alpha = \frac{1}{2}$ (procédé dichotomique). De plus, la complexité $P(n)$ du partitionnement et la complexité $F(n)$ de la fusion sont souvent polynômiales donc il existe $\gamma \in \mathbb{R}_+$ tel que $P(n) = O(n^\gamma)$ et $F(n) = O(n^\gamma)$. Nous obtenons alors l'équation de complexité :

$$C(n) = qC(n/2) + O(n^\gamma)$$

On peut alors appliquer le résultat suivant démontré au chapitre 3 :

Théorème 1 (Complexité des algorithmes "diviser pour régner")

Soit $q \in \mathbb{N}^*$, $\gamma \in \mathbb{R}_+$ et l'équation de complexité :

$$C(n) = qC(n/2) + O(n^\gamma)$$

- Si $\log_2(q) = \gamma$, alors $C(n) = O(n^\gamma \log_2(n))$.
- Si $\log_2(q) > \gamma$, alors $C(n) = O(n^{\log_2(q)})$.
- Si $\log_2(q) < \gamma$, alors $C(n) = O(n^\gamma)$.

2 Un premier exemple : l'exponentiation rapide

Soit $n \in \mathbb{N}$ et soit E un ensemble muni d'une loi $*$ associative et possédant un élément neutre noté e. Nous cherchons à calculer, pour tout $x \in E$, x^n.

2.1 Algorithme naif

Version itérative

```
let puissance x n =
  let p = ref 1 in
    for k = 1 to n do
      p := !p * x
    done;
  !p
;;
```

Complexité : En choisissant n comme taille de données et la multiplication comme opération fondamentale, on obtient une complexité linéaire $C(n) = n$.

Version récursive

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

Complexité : Avec les mêmes choix pour la complexité,

$$\begin{cases} C(0) = 0 \\ \forall n \in \mathbb{N}^*,\ C(n) = 1 + C(n-1) \end{cases}$$

Comme $(C(n))_{n\in\mathbb{N}}$ est une suite arithmétique de raison 1 et de premier terme 0, on obtient une complexité linéaire $C(n) = n$.

2.2 Algorithme d'exponentiation rapide

On applique le paradigme "diviser pour régner" : pour manipuler un objet de taille n, on se ramène à plusieurs objets de taille strictement inférieure à n. Dans le cas du calcul de x^n :

$$x^n = \begin{cases} (x^{n/2})^2 & \text{si } n \text{ pair} \\ (x^{(n-1)/2})^2 * x & \text{si } n \text{ impair} \end{cases}$$

avec

- en vert le partitionnement en deux objets de taille strictement inférieure à n,
- en rouge le fusionnement des résultats obtenus pour obtenir x^n.

Version récursive

```
let rec exponentiation_rapide x n = match n with
  | 0 -> 1
  | _ -> if n mod 2 = 0
         then (exponentiation_rapide x (n/2)) * (exponentiation_rapide x (n/2))
         else (exponentiation_rapide x (n/2)) * (exponentiation_rapide x (n/2)) * x
;;
```

Complexité : En choisissant n comme taille de données et la multiplication comme opération fondamentale,

$$\begin{cases} C(0) = 0 \\ \forall n \in \mathbb{N}^*,\ C(n) = 2C(n/2) + 2. \end{cases}$$

En appliquant le théorème 1, avec $q = 2$ et $\gamma = 0$, on obtient : $C(n) = O(n^{\log_2(q)}) = O(n)$. La complexité est linéaire et on ne l'a donc pas améliorée. On s'y est mal pris en faisant deux appels récursifs au lieu d'un.

```
let rec exponentiation_rapide x n = match n with
  | 0 -> 1
  | _ -> let aux = exponentiation_rapide x (n/2) in
          if n mod 2 = 0 then aux*aux
                         else aux*aux*x
;;
```

Complexité : Avec les mêmes choix pour la complexité,

$$\begin{cases} C(0) = 0 \\ \forall n \in \mathbb{N}^*,\ C(n) = C(n/2) + 2. \end{cases}$$

En appliquant le théorème 1, avec $q = 1$ et $\gamma = 0$, on obtient : $C(n) = O(n^\gamma \log_2(n)) = O(\log_2(n))$. La complexité est logarithmique.

Version itérative

Commençons par proposer une version récursive terminale. Remarquons que :

$$x^n = \begin{cases} (x^{n/2})^2 & \text{si } n \text{ pair} \\ x * (x^{(n-1)/2})^2 & \text{si } n \text{ impair} \end{cases} = \begin{cases} (x^2)^{n/2} & \text{si } n \text{ pair} \\ x * (x^2)^{(n-1)/2} & \text{si } n \text{ impair} \end{cases}$$

Nous somme de nouveau ramené au calcul de x^n avec "$x = x * x$", "$n = \lfloor n/2 \rfloor$" et "$acc = acc * x$" si n impair. Testons pour $n = 13$:

n	x	acc
13	x	1
6	x^2	$x \times 1$
3	x^4	$x \times 1$
1	x^8	$x^4 \times x \times 1$
0	x^{16}	$x^8 \times x^4 \times x \times 1$

On obtient la version récursive terminale suivante :

```
let exponentiation_rapide_terminale x n =
  let rec exponentiation_rapide_aux x n acc = math n with
    | 0 -> acc
    | n when n mod 2 = 0 -> exponentiation_rapide_aux (x*x) (n/2) acc
    | _ -> exponentiation_rapide_aux (x*x) (n/2) (acc*x)
  in exponentiation_rapide_aux x n 1
;;
```

On en déduit une version itérative :

```
let exponentiation_rapide_iter x n =
  let acc = ref 1 and nn = ref n and xx = ref x in
    while (!nn <> 0) do
      if (!nn mod 2 = 1) then acc := (!acc)*(!xx);
      xx := (!xx)*(!xx);
      nn := (!nn)/2
    done;
  !acc
;;
```

Complexité : En faisant les divisions euclidiennes de 13 par 2, on obtient l'écriture binaire de 13 :

$$13 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = \overline{1101}^2.$$

puis

$$x^{13} = x^{1\times 2^3+1\times 2^2+0\times 2^1+1\times 2^0} = \left(x^{2^3}\right)^1 \times \left(x^{2^2}\right)^1 \times \left(x^{2^1}\right)^0 \times \left(x^{2^0}\right)^1.$$

De manière plus générale, $n \in \mathbb{N}^*$ a pour écriture binaire :

$$n = a_p 2^p + a_{p-1}2^{p-1} + \ldots + a_2 2^2 + a_1 2^1 + a_0 2^0 = \overline{a_p a_{p-1} \ldots a_2 a_1 a_0}^2,$$

où $p \in \mathbb{N}$, $a_0, a_1, a_2, \ldots a_{p-1} \in \{0, 1\}$ et $a_p = 1$. Les $a_0, a_1, a_2, \ldots, a_{p-1}, a_p$ sont les restes des divisions successives de n par 2.
Nous avons $(p+1)$ itérations à effectuer avec à chaque fois 2 multiplications soit un total de $2(p+1)$ multiplications. Et :

$$2^p \leq n \leq \underbrace{2^p + 2^{p-1} + \ldots + 2^2 + 2^1 + 2^0}_{=2^{p+1}-1} \qquad \text{donc} \qquad 2^p \leq n < 2^{p+1}.$$

Par croissance de la fonction logarithme, $p \leq \log_2(n) < p+1$ et donc $p = \lfloor \log_2(n) \rfloor$. On obtient une complexité logarithmique $C(n) = O(\log_2(n))$.

2.3 Application à la suite de Fibonacci

Considérons la suite de Fibonacci définie par :

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ \forall n \in \mathbb{N},\ F_{n+2} = F_{n+1} + F_n. \end{cases}$$

Version itérative

```
let fib n =
  let a = ref 0 and b = ref 1 and aux = ref 0 in
    for k = 1 to n do
      aux := !b;
      b := !a + !b;
      a := !aux
    done;
  !a
;;
```

Complexité : En choisissant n comme taille de données, l'addition comme opération fondamentale, on obtient une complexité linéaire $C(n) = n$.

Version récursive naïve

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

Complexité : Avec les mêmes choix pour la complexité,

$$\begin{cases} C(0) = C(1) = 0 \\ \forall n \in \mathbb{N} \setminus \{0,1\},\ C(n) = C(n-1) + C(n-2) + 1 \end{cases}$$

Introduisons la suite $(u_n)_{n\in\mathbb{N}}$ définie par : $\forall n \in \mathbb{N},\ u_n = C(n) + 1$. Alors $u_0 = 1$, $u_1 = 1$ et pour tout $n \in \mathbb{N} \setminus \{0,1\}$,

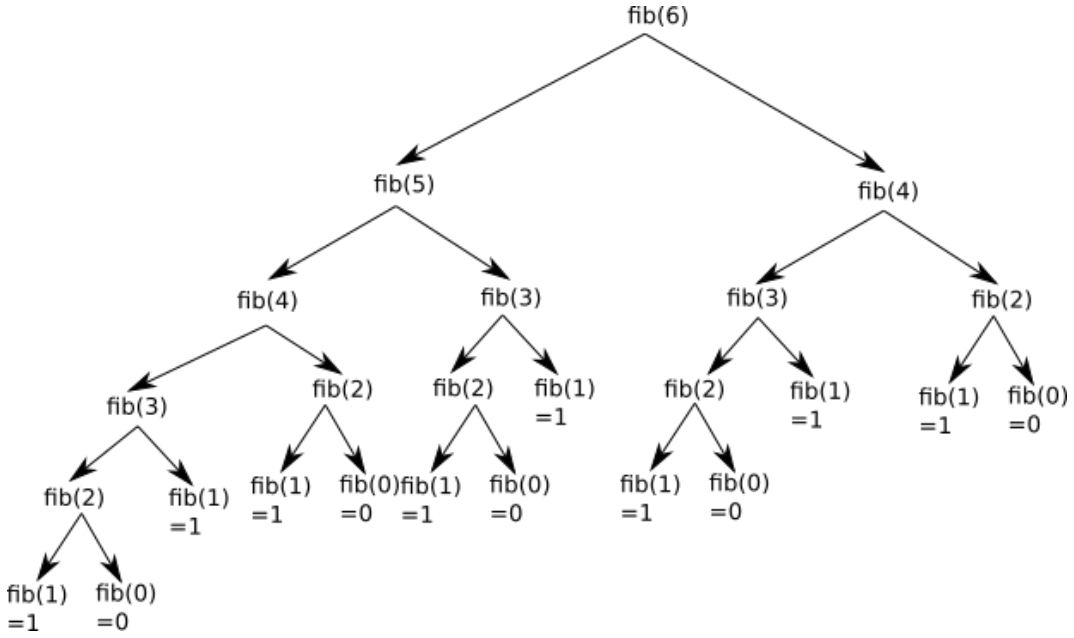
$$u_n = C(n) + 1 = C(n-1) + C(n-2) + 1 + 1 = u_{n-1} - 1 + u_{n-2} - 1 + 1 + 1 = u_{n-1} + u_{n-2}.$$

Donc $(u_n)_{n\in\mathbb{N}}$ est une suite récurrente linéaire d'ordre 2 (c'est la suite de Fibonacci, avec un décalage d'indice de 1) et on obtient :

$$\forall n \in \mathbb{N},\ u_n = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^{n+1}$$

Donc $C(n) = u_n - 1 = O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$. La complexité est exponentielle !

Cette version récursive naïve est loin d'être efficace. Voici par exemple l'arbre d'appels pour le calcul de F_6 par cette méthode :



On remarque que les mêmes termes sont recalculés plusieurs fois. En gardant en mémoire les différents termes déjà calculés, on évite des calculs inutiles.

Version récursive avec mémoïsation

La mémoïsation consiste à conserver dans une table les différents résultats calculés au cours des différents appels récursifs.

```
let rec fib_aux n table =
  if (table.(n) = -1) then
     table.(n) <- (fib_aux (n-1) table) + (fib_aux (n-2) table);
  table.(n)
;;

let fib n=
   let table = Array.make (n+1) -1 in
      table.(0) <- 0;
      table.(1) <- 1;
      fib_aux n table
;;
```

Complexité : En choisissant n comme taille de données, l'addition comme opération fondamentale, on a :

$$\begin{cases} C(0) = C(1) = 0 \\ \forall n \in \mathbb{N} \setminus \{0,1\},\ C(n) = 1 + C(n-1) \end{cases}$$

Comme $(C(n))_{n\in\mathbb{N}^*}$ est une suite arithmétique de raison 1 et de premier terme 0, on obtient $C(n) = n-1$ donc $C(n) \sim n$. La complexité est linéaire.

Version avec utilisation de l'exponentation rapide

L'algorithme d'exponentiation rapide s'applique dans tous les ensembles E muni d'une loi associative et possédant un élément neutre, comme $\mathbb{K}[X]$ muni du produit polynomiale ou $\mathcal{M}_n(\mathbb{K})$ muni du produit matriciel.

Notons, pour tout $n \in \mathbb{N}$, $X_n = \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix}$ et $A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$. Alors :

$$AX_n = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} F_{n+1} \\ F_n + F_{n+1} \end{pmatrix} = \begin{pmatrix} F_{n+1} \\ F_{n+2} \end{pmatrix} = X_{n+1}.$$

Par analogie aux suites géométriques, on obtient :

$$\forall n \in \mathbb{N},\ X_n = A^n X_0.$$

Ainsi, le calcul de F_n est ramener au calcul de X_n et par suite au calcul de A^n.

Si $A = \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix}$ et $B = \begin{pmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{pmatrix}$ sont deux matrices de $\mathcal{M}_2(\mathbb{K})$,

$$A \times B = \begin{pmatrix} a_{1,1}b_{1,1} + a_{1,2}b_{2,1} & a_{1,1}b_{1,2} + a_{1,2}b_{2,2} \\ a_{2,1}b_{1,1} + a_{2,2}b_{2,1} & a_{2,1}b_{1,2} + a_{2,2}b_{2,2} \end{pmatrix}.$$

Le produit matriciel a un coût de 12 opérations fondamentales, soit un coût constant en $O(1)$. Le calcul de A^n par l'algorithme d'exponentiation rapide nous conduit à un coût de $12\lfloor \log_2(n) \rfloor$ opérations fondamentales. On obtient ainsi une complexité logarithmique.

Exercice 1

L'objectif de cet exercice est de programmer sur OCaml la méthode présentée ci-dessus.

1. Définir une fonction `produit_matriciel` qui calcule le produit de $A \in \mathcal{M}_{n,p}(\mathbb{N})$ et $B \in \mathcal{M}_{q,r}(\mathbb{N})$ si $p = q$ et retourne un message d'erreur sinon.
2. En déduire une fonction `fib` qui calcule le terme F_n en utilisant la méthode de l'exponentiation rapide.

Exercice 2

On considère toujours la suite de Fibonacci définie par $F_0 = 0$, $F_1 = 1$ et pour tout $n \in \mathbb{N}$, $F_{n+2} = F_{n+1} + F_n$.

1. Montrer que pour tout couple $(p,q) \in \mathbb{N} \times \mathbb{N}^*$, $F_{p+q} = F_{p+1}F_q + F_pF_{q-1}$.
2. En déduire une fonction `fib : int -> int` qui calcule le terme F_n en $O(\log_2(n))$ opérations arithmétiques. On pourra distinguer les cas selon la parité de n.

---

3 D'autres exemples d'applications du DpR

3.1 Recherche d'un élément dans un tableau trié

Soit $(E, \leq)$ un ensemble ordonné, $n \in \mathbb{N}^*$, s une série de n éléments de E triées par ordre croissant et $x \in E$. Nous étudions l'appartenance de x à s. La série s est implémentée en OCaml sous la forme d'un tableau.

Algorithme naif

On n'utilise pas ici le fait que la série soit triée.

• En itératif :

```
let recherche s x =
  let n = Array.length s and i = ref 0 in
    while (!i < n) && (s.(!i) <> x) do
      i := !i + 1
    done;
    !i <> n
;;
```

• En récursif :

```
let rec recherche_aux s x i = match i with
  | -1 -> False
  | _ -> (x = s.(i)) || (recherche_aux s x (i - 1))
;;

let recherche s x = recherche_aux s x (Array.length s - 1);;
```

Complexité : En choisissant n comme taille de données, la comparaison comme opération fondamentale, on obtient une complexité linéaire $C(n) = O(n)$.

En appliquant le DpR

On applique le paradigme "diviser pour régner". On compare x et $s_{n/2}$. Si $x \leq s_{n/2}$, on poursuit la recherche dans la première moitié du tableau et sinon, dans la seconde moitié. Lorsque la recherche s'effectue dans un tableau ne contenant qu'un élément, il suffit de tester si cet élément est égal à x.

• En itératif :

```
let recherche s x =
  let g = ref 0 and d = ref (Array.length s - 1) in
    while (!g < !d) do
      let m = (!g + !d) / 2 in
        if (x <= s.(m)) then d := m
                        else g := m + 1
    done;
    x = s.(!d)
;;
```

• En récursif :

Afin d'améliorer la complexité spatiale, nous allons travailler en place, c'est-à-dire uniquement sur les indices du tableau s et non sur des sous-tableaux de s (en utilisant la commande `Array.sub`).

```
let recherche s x =
  let rec recherche_aux s x g d =
    if g = d then x = s.(d)
             else let m = (g + d) / 2 in
                    if x <= s.(m) then recherche_aux s x g m
                                  else recherche_aux s x (m + 1) d
  in recherche_aux s x 0 (Array.length s - 1)
;;
```

Complexité : Toujours avec les mêmes choix pour la complexité,

$$\begin{cases} C(0) = 0 \\ \forall n \in \mathbb{N}^*,\ C(n) = C(n/2) + 1 \end{cases}$$

En appliquant le théorème 1, avec $q = 1$ et $\gamma = 0$, on obtient $C(n) = O(\log_2(n))$. La complexité est logarithmique.

Terminaison : Supposons $g < d$. Nous effectuons un appel récursif sur (g, m) ou sur $(m+1, d)$ avec $m = \left\lfloor \frac{g+d}{2} \right\rfloor$.
Par définition de la partie entière,

$$\frac{g+d}{2} - 1 < m \leq \frac{g+d}{2}.$$

Alors :

- $m - g \leq \frac{g+d}{2} - g = \frac{d-g}{2} < d - g.$
- $d - (m+1) = d - 1 - m < d - 1 - \frac{g+d}{2} + 1 = \frac{d-g}{2} < d - g.$

Ainsi, la suite $(d - g)$ est une suite strictement décroissante d'entiers naturels. Donc l'algorithme se termine.

3.2 Algorithme de Karatsuba

Nous choisissons de travailler avec des polynômes de $\mathbb{Z}[X]$ représentés par des tableaux de leurs coefficients (l'indice i correspond au coefficient du monôme de degré i). Nous travaillerons donc avec des données de types `Array` en OCaml.

Somme de deux polynômes

Nous supposons ici les polynômes de même longueur n.

```
let addition a b =
  let n = Array.length a in
    let c = Array.make n a.(0) in
      for k = 0 to (n-1) do
        c.(k) <- a.(k) + b.(k)
      done;
    c
;;
```

Complexité : En choisissant n comme taille de données, les opérations arithmétiques comme opérations fondamentales, on obtient une complexité linéaire $C(n) = n$.

Produit naïf de deux polynômes

Nous supposons encore les polynômes de même longueur n.

```
let multiplication a b =
  let n = Array.length a in
    let c = Array.make (2*n-1) 0 in
      for j = 0 to (n-1) do
        for i = 0 to (n-1) do
          c.(i + j) <- c.(i + j) + a.(i) * b.(j)
        done;
      done;
    c
;;
```

Complexité : Toujours avec les mêmes choix pour la complexité, on a une complexité quadratique $C(n) = O(n^2)$.

Produit de deux polynômes en appliquant le DpR

Cherchons à améliorer la complexité à l'aide du paradigme "diviser pour régner". Afin de simplifier, nous supposons désormais que n est de la forme 2^p, avec $p \in \mathbb{N}$. Ainsi, A et B étant deux polynômes de degré strictement inférieur à n,

$$\begin{aligned} A(X) &= \underbrace{a_0 + a_1 X + \ldots + a_{n/2-1}X^{n/2-1}}_{n/2 \text{ coefficients}} + \underbrace{a_{n/2}X^{n/2} + a_{n/2+1}X^{n/2+1} + \ldots + a_{n-1}X^{n-1}}_{n/2 \text{ coefficients}} \\ &= \underbrace{a_0 + a_1 X + \ldots + a_{n/2-1}X^{n/2-1}}_{=:A_1(X)} + X^{n/2} \underbrace{\left(a_{n/2} + a_{n/2+1}X \ldots + a_{n-1}X^{n/2-1}\right)}_{=:A_2(X)} \end{aligned}$$

où A_1 et A_2 sont deux polynômes de degré strictement inférieur à $n/2$. On introduit de même B_1 et B_2 tels que $B(X) = B_1(X) + X^{n/2}B_2(X)$. Alors :

$$\begin{aligned} A(X) \times B(X) &= \left(A_1(X) + X^{n/2}A_2(X)\right) \times \left(B_1(X) + X^{n/2}B_2(X)\right) \\ &= A_1(X) \times B_1(X) + X^{n/2}(A_1(X) \times B_2(X) + A_2(X) \times B_1(X)) + X^n(A_2(X) \times B_2(X)). \end{aligned}$$

Dans ce calcul,

- le partitionnement correspond à l'obtention de A_1, A_2, B_1, B_2 ce qui ne nécessite pas d'opérations algébriques,
- les multiplications par $X^{n/2}$ et X^n seront interprétées comme des décalages,
- la fusion correspond à l'addition des différents produits et est de complexité linéaire.

L'équation de complexité s'écrit :

$$C(n) = P(n) + 4C(n/2) + F(n) = 4C(n/2) + O(n).$$

En appliquant le théorème 1, avec $q = 4$ et $\gamma = 1$, on obtient $C(n) = O(n^2)$. On n'a pas réussi à améliorer la complexité...

L'idée de Karatsuba, c'est de faire seulement trois produits :

$$A_1 \times B_1, \qquad A_2 \times B_2 \qquad \text{et} \qquad (A_1 + A_2) \times (B_1 + B_2) - \underbrace{(A_1 \times B_1 + A_2 \times B_2)}_{\text{déjà calculé}} = A_1 \times B_2 + A_2 \times B_1.$$

L'équation de complexité s'écrit alors :

$$C(n) = 3C(n/2) + O(n)$$

En appliquant le théorème 1, avec $q = 3$ et $\gamma = 1$, on obtient $C(n) = O(n^{\log_2(3)}) \simeq O(n^{1,585})$. C'est mieux que la complexité quadratique obtenue avec l'algorithme naïf.

Exercice 3

L'objectif de cet exercice est de programmer sur OCaml l'algorithme de Karatsuba.

1. Construire une fonction `fusion` qui, étant donné trois polynômes A, B, C de degré strictement inférieur à n, calcule $A + X^{n/2}B + X^nC$.
2. En déduire une fonction `karatsuba` qui calcule le produit de deux polynômes en utilisant la méthode de Karatsuba.

Exercice 4

Une paire $\{i, j\} \in [\![1, n]\!]^2$ est une inversion pour la permutation $\sigma \in S_n$ si $(i-j)(\sigma(i) - \sigma(j)) < 0$. Par exemple, la permutation suivante admet 4 inversions :

$$\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 2 & 1 & 3 & 5 & 7 & 4 & 6 \end{pmatrix}.$$

Déterminer une fonction, utilisant le paradigme "diviser pour régner", de complexité quasi-linéaire qui renvoie le nombre d'inversion d'une permutation donnée par le tableau d'entiers de ses images.
