Diviser et Régner

I. L'algorithme de Karatsuba pour les polynômes

Dans la suite, on représentera les polynômes par des tableaux d'entiers. On pourra définir un type `poly` à l'aide de la ligne :

```
type poly = int array ;;
```

1. Quelques fonctions de base

Question 1 Écrire une fonction `somme: poly -> poly -> poly` qui renvoie la somme de deux polynômes.

Question 2 Écrire une fonction `produit_naif: poly -> poly -> poly` qui renvoie le produit de deux polynômes, calculé « naïvement. »

2. L'algorithme de Karatsuba

On peut facilement adapter l'algorithme de Karatsuba à la multiplication de polynômes. Ainsi, en écrivant

$$P = AX^n + B \qquad Q = CX^n + D$$

avec $A, B, C, D \in \mathbf{K}_n[X]$, alors

$$PQ = ACX^{2n} + \big((A+B)(C+D) - AC - BD\big)X^n + BD$$

Ainsi, pour multiplier deux polynômes de degrés au plus $2n$, on se ramène à trois (et non quatre) produits de polynômes de degrés au plus n, ce qui permet d'avoir une complexité $n^{\log_2 3}$ au lieu du n^2 de l'algorithme naïf.

Question 3 Écrire une fonction

```
sous_tableau: 'a array -> int -> int -> 'a array
```

telle que `sous_tableau t deb fin` retourne un tableau de longueur fin – deb constitué des valeurs correspondantes du tableau passé en argument.

```
# sous_tableau [| 2; 4; 1; 3; 6; 5 |] 2 5 ;;
- : int array = [| 1; 3; 6 |]
```

Avec les notations précédentes, cette fonction pourra servir, par exemple, à extraire les polynômes A et B à partir de P.

Question 4 Après avoir lu la question **en entier**, écrire une fonction

```
kara: poly -> poly -> poly
```

qui calcule le produit de deux polynômes. On supposera que les deux polynômes passés en argument sont représentés par des tableaux de même taille, laquelle taille est une puissance de 2, et que le polynôme retourné est représenté par un tableau de taille double.

Question 5 En déduire une fonction `karatsuba: poly -> poly -> poly` qui calcule le produit de deux polynômes représentés par des tableaux de tailles *quelconques*.

Question 6 Tester et comparer les fonctions `produit_naif` et `karatsuba`. En particulier, on pourra écrire une fonction qui engendre des tableaux de valeurs aléatoires, vérifier que les méthodes de calcul de produits ont des résultats égaux, et on pourra comparer les temps d'éxecution pour des tailees de polynômes différentes.

II. Le nombre d'inversions d'un tableau

Étant donné un tableau d'entier t de taille n, une **inversion** de t est un couple d'entiers $(i, j) \in [\![0, n-1]\!]^2$ tel que $i < j$ et $t_i > t_j$.

On cherche à déterminer le nombre d'inversions d'un tableau donné.

Question 7 Écrire une fonction **non récursive**

```
inversions_naif : int array -> int
```

qui calcule le nombre d'inversions du tableau passé en argument. Quelle est la complexité de cette fonction ?

Nous allons maintenant essayer de procéder de façon plus efficace. Pour cela, nous allons faire un détour par des listes triées.

Question 8 Étant donné deux listes d'entiers `l1` et `l2` triées par ordre croissant, déterminer comment déterminer facilement et efficacement le cardinal de l'ensemble $\{(u, v) \in \texttt{l1} \times \texttt{l2} \mid u > v\}$?

Par exemple, pour `l1 = [2; 4; 5; 14; 16]` et `l2 = [3; 7; 8; 15; 17]`, ce cardinal vaut 9.

Question 9 En s'inspirant de ce que vous venez de faire à la main, écrire une fonction

```
pre_inversions : 'a list -> 'a list -> int
```

qui, étant donné deux listes triées par ordre croissant, renvoie le cardinal précédent. On veillera à avoir une complexité linéaire en la somme des longueurs des listes.

Ainsi, on veut :

```
# pre_inversions [2; 4; 5; 14; 16] [3; 7; 8; 15; 17] ;;
- : int = 9
# pre_inversions [3; 7; 8; 15; 17] [2; 4; 5; 14; 16] ;;
- : int = 16
```

Question 10 En s'inspirant de l'exercice *Buy and Sell*, en déduire une fonction :

```
inversions: 'a list -> int
```

qui calcule le nombre d'inversions d'un tableau à l'aide d'une approche « Diviser pour Régner ». Quelle est sa complexité ?

Question 11 Comparer les résultats et temps d'exécution des deux fonctions.