Devoir en temps limité du Vendredi 2 Mai

Dans l'ensemble du devoir, toutes les fonctions seront :

- *écrites en langage OCaml,*
- *précédées d'une explication des variables utilisées,*
- *précédées d'une explication de l'algorithme.*

Exercice 1 (Nombre d'inversions d'une permutation)

1. (a) Écrire une fonction `fusion t d m f` qui, le tableau t vérifiant

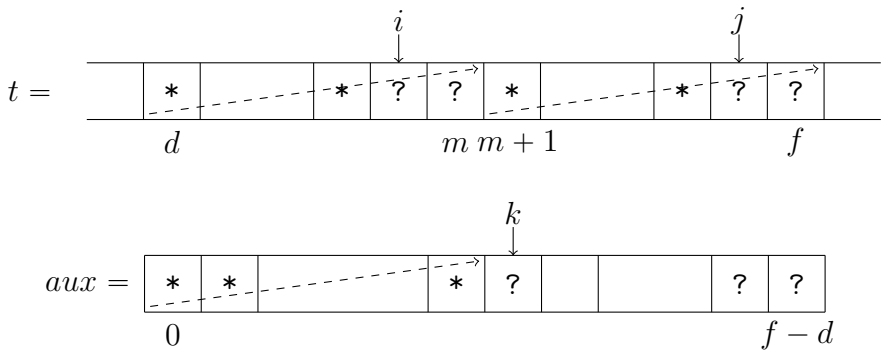
$$\begin{cases} t.(d) \leqslant t.(d+1) \leqslant \ldots \leqslant t.(m-1) \leqslant t.(m) \\ t.(m+1) \leqslant t.(m+2) \leqslant \ldots \leqslant t.(f-1) \leqslant t.(f) \end{cases},$$

le modifie de telle façon à ce qu'il vérifie :

$$t.(d) \leqslant t.(d+1) \leqslant \ldots \leqslant t.(m) \leqslant \ldots \leqslant t.(f-1) \leqslant t.(f).$$

L'idée peut s'illustrer de la manière suivante : vous avez deux paquets de cartes triés, les plus petites sur le dessus, chaque étape élémentaire consiste à choisir la carte la plus petite parmi les deux au sommet de chaque paquet et à la placer sur le paquet trié en construction.

Plus formellement, nous respecterons l'invariant de boucle suivant :



où :

$$\begin{cases} \{aux.(p)\,,\, p \in [\![0, k-1]\!]\} = \{t.(p)\,,\, p \in [\![d, i-1]\!]\} \cup \{t.(p)\,,\, p \in [\![m, j-1]\!]\} \\ aux.(0) \leqslant aux.(1) \leqslant ... \leqslant aux.(k-1) \end{cases},$$

et nous terminerons en recopiant le tableau aux dans le tableau t.

(b) Le tri fusion peut se décrire avec le paradigme "diviser pour régner" de la manière suivante :

— **Partition** : Nous divisons la série des n données à trier en deux séries de $n/2$ données.

— **Tri récursif** : Nous trions ensuite les deux séries récursivement.

— **Fusion** : Nous terminons en fusionnant les deux séries triées.

Écrire une fonction en OCaml `tri_fusion t` qui trie le tableau t d'après l'algorithme du tri fusion.

2. On appelle **permutation** de $[\![1;n]\!]$ toute bijection σ de $[\![1;n]\!]$ sur lui-même.
   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 :

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

   (a) Modifier la fonction `fusion t d m f` pour obtenir une fonction `fusion_et_inversions t d m f`, qui, en plus d'effectuer la fusion des deux sous-tableaux triés, compte et renvoit le nombre d'inversions entre ces deux sous-tableaux ; c'est-à-dire, pour chaque élément du second sous-tableau, compte le nombre d'éléments du premier sous-tableau qui lui sont supérieurs.

   (b) En déduire une fonction `inversions t`, obtenue en modifiant la fonction `tri_fusion t`, qui renvoie le nombre d'inversions d'une permutation donnée par le tableau d'entiers de ses images (et y opère un tri fusion), en utilisant le paradigme "diviser pour régner".
   Quelle est la complexité de cette fonction ?

Exercice 2 (Nombre d'occurrences d'un élément dans un ensemble)

Nous est donné un nombre fini d'éléments d'un ensemble E, n désigne le nombre d'éléments donnés et nous noterons $a_0, a_1, \ldots, a_{n-1}$ ces éléments.
Pour un élément x appartenant à E, nous recherchons le nombre de valeurs parmi les valeurs $a_0, a_1, \ldots, a_{n-1}$ égales à x.

1. Une approche itérative.
   Les éléments $a_0, a_1, \ldots, a_{n-1}$ sont stockés dans un tableau `a`.

   (a) Écrire en OCaml, une fonction itérative `occurrences1 a x` qui a pour argument une variable `a` de type `Array` et une variable `x` de type identique à celui des éléments contenus dans le tableau `a`, et renvoie le nombre de valeurs parmi les valeurs contenues dans le tableau `a` égales à `x`.

   (b) Prouver la correction de cet algorithme.

   (c) Calculer la complexité temporelle de cet algorithme.

2. Une approche récursive.
   Les éléments $a_0, a_1, \ldots, a_{n-1}$ sont stockés dans une liste `a`.

   (a) Écrire en OCaml, une fonction récursive (non terminale) `occurences2 a x` qui a pour argument une variable `a` de type `List` et une variable `x` de type identique à celui des éléments contenus dans la liste `a`, et renvoie le nombre de valeurs parmi les valeurs contenues dans la liste `a` égales à `x`.

   (b) Prouver la terminaison de cet algorithme.

   (c) Déterminer la complexité temporelle de cet algorithme.

3. Une deuxième approche récursive.
   Proposer une version récursive terminale de la fonction `occurences2`.

Exercice 3 (Karatsuba avec des listes de flottants)

On représente un polynôme appartenant à $\mathbb{R}[X]$ sous la forme de la liste de ses coefficients, la tête correspondant au degré zéro.
Par exemple, la liste `3. :: 4. :: 5. :: []` correspond au polynôme $3,0+4,0X+5,0X^2$.

1. Opérations préliminaires :
   (a) Écrire une fonction récursive `normalise` qui retire les zéros inutiles en fin de liste de signature :
   ```
   float list -> float list
   ```
   (b) Écrire une fonction récursive `ajout` qui fait la somme de deux polynômes de signature :
   ```
   float list -> float list -> float list
   ```
   (c) Écrire une fonction récursive `mulscal` qui fait la multiplication d'un polynôme par un scalaire de signature :
   ```
   float -> float list -> float list
   ```
   (d) Écrire une fonction récursive `soustr` qui fait la différence de deux polynômes de signature :
   ```
   float list -> float list -> float list
   ```
2. Multiplication naïve :
   La multiplication de deux polynômes $P = \sum_k a_k X^k$ et $Q = \sum_k b_k X^k$ est donnée par la formule :
   $$PQ = \sum_k \left(\sum_{i+j=k} a_i b_j \right) X^k$$
   La représentation des polynômes sous forme de liste rend cette formule rédhibitoire (car le simple accès à un coefficient quelconque du polynôme se fait en temps linéaire).
   Cependant, on sait accéder en temps constant au coefficient de degré zéro. On peut donc écrire : $P = a_0 + P_1 X$ et $Q = b_0 + Q_1 X$ avec P_1 et Q_1 deux polynômes.
   À l'aide du développement de $(a_0 + P_1 X)(b_0 + Q_1 X)$, écrire une fonction récursive `mult` de signature :
   ```
   float list -> float list-> float list
   ```
   qui renvoie le produit de deux polynômes.
3. Algorithme de Karatsuba :
   On s'intéresse ici à une autre façon de décomposer un polynôme. Pour tout entier k, si on a $P = RX^k + S$ et $Q = TX^k + U$, alors le produit s'écrit :
   $$PQ = X^{2k}RT + X^k(RU + ST) + SU$$
   En négligeant la multiplication par un X^j, cette méthode naïve requiert quatre multiplications de polynômes plus petits (si on impose les degrés de S et U strictement inférieurs à k). Or, Karatsuba (1960) a remarqué que, si on écrit ce produit sous la forme :
   $$PQ = X^{2k}RT + X^k((R+S)(T+U) - RT - SU) + SU$$
   alors le produit ne requiert que trois multiplications (au lieu de quatre naïvement) de polynômes plus petits : RT, SU et $(R+S)(T+U)$.
   Cette remarque ouvre la voie à une implémentation en diviser pour régner.

(a) Écrire la fonction `separe l k` de signature

```
float list -> int -> float list * float list
```

qui renvoie le couple `(l1, l2)` où `l1` est la liste des k premiers termes de `l`, et `l2` le reste. Par exemple :

```
separe [1.;2.;3.;4.;5.;6.;7.;8.] 3;;
- : float list * float list = ([1.; 2.; 3.], [4.; 5.; 6.; 7.; 8.])
```

(b) Écrire une fonction `produit_x` de signature

```
int -> float list -> float list
```

qui, étant donné un entier i et un polynôme P, renvoie le polynôme X^iP.

(c) Écrire une fonction récursive `karatsuba` de signature

```
int -> float list -> float list -> float list
```

qui, étant donné un entier k et deux polynômes P et Q, renvoie le produit $P \times Q$ suivant une méthode diviser pour régner avec le paramètre k pour la décomposition de deux polynômes. On pourra supposer que k est une puissance de 2.
