DS3 : 10 décembre 2022 – 4h00

Pour chaque question, le langage imposé par l'énoncé doit être respecté.

Les fonctions doivent être documentées et commentées à bon escient. Le code doit être indenté et d'un seul tenant (pas de flèche pour dire qu'un bout de code se situe ailleurs par exemple). Le code illisible ne sera pas corrigé.

Les réponses doivent être justifiées et les résultats encadrés, sans quoi la question correspondante ne sera pas corrigée.

Pour le code en C, on suppose que les fichiers d'en-tête `stdbool.h`, `stdint.h` et `stdio.h` sont inclus.

Tout document interdit. Calculatrice interdite. Téléphone éteint et rangé dans le sac.

Ce qui est illisible ou trop sale ne sera pas corrigé.
Soulignez ou encadrez vos résultats.

Complexité

Lorsque la complexité temporelle d'une fonction f dépend de plusieurs paramètres m et n, on dira que f a une complexité en $\mathcal{O}(\phi(m,n))$ lorsqu'il existe trois constantes c, M et N telles que la complexité de f est inférieure ou égale à $c\,\phi(m,n)$ pour tout $m > M$ et $n > N$.

1 Étude des partitions non croisées

Le langage de programmation utilisé dans cette partie est le langage `OCaml`.

Dans cette partie, on introduit et on étudie de façon élementaire les partitions non croisées, objets combinatoires apparaissant dans divers domaines des mathématiques, notamment dans la théorie des probabilités libres et des matrices aléatoires.

Dans la suite, pour tout couple $(i,j) \in \mathbb{N}^2$, les notations $[\![n]\!]$ et $[\![i,n]\!]$ désignent respectivement les ensembles $\{1,2,\dots,n\}$ et $\{i,i+1,\dots,n\}$. Par convention, $[\![0]\!] = \emptyset$ et si $i > n$, alors $[\![i,n]\!] = \emptyset$.

Définition — Partition. Soit A un sous-ensemble non vide de $\mathbb{N}$. Une *partition* $\mathscr{P}$ de A est un ensemble de parties de A tel que :

1. $\forall P \in \mathscr{P}, P \neq \emptyset$; 2. $\forall (P,Q) \in \mathscr{P}^2, (P \neq Q) \Longrightarrow (P \cap Q = \emptyset)$; 3. $\bigcup_{P \in \mathscr{P}} = A$.

Par exemple, $\{\{2,4,5\},\{7,9\},\{8\}\}$ est une partition de l'ensemble $\{2,4,5,7,8,9\}$.

Définition — Classe. Soit $\mathscr{P}$ une partition d'un sous-ensemble A non vide de $\mathbb{N}$. Soit i un élément de A. La *classe* de i est l'unique élément de $P \in \mathscr{P}$ tel que $i \in P$. Elle est notée $\texttt{Cl}(i)$.

Par exemple, pour $\mathscr{P} = \{\{2,4,5\},\{7,9\},\{8\}\}$, on a $\texttt{Cl}(4) = \{2,4,5\}$.

Définition — Partition non croisée. Soit $\mathscr{P}$ un partition d'un sous-ensemble A non vide de $\mathbb{N}$. On dit que $\mathscr{P}$ est *non croisée* si pour tout quadruplet $(a,b,c,d) \in A^4$ tel que $a < b < c < d$ on a :

$$\big(\texttt{Cl}(a) = \texttt{Cl}(c) \text{ et } \texttt{Cl}(b) = \texttt{Cl}(d)\big) \quad \Longrightarrow \quad \big(\texttt{Cl}(a) = \texttt{Cl}(b) = \texttt{Cl}(c) = \texttt{Cl}(d)\big).$$

Par exemple, $\mathscr{P} = \{\{2,4,5\},\{7,9\},\{8\}\}$ est bien une partition non croisée de $\{2,4,5,7,8,9\}$. En effet, aucun quadruplet $(a,b,c,d) \in \{2,4,5,7,8,9\}^4$ ne vérifie la condition $\texttt{Cl}(a) = \texttt{Cl}(c)$ et $\texttt{Cl}(b) = \texttt{Cl}(d)$.

De même, $\mathscr{Q} = \{\{2,7,8,9\},\{4,5\}\}$ est une partition non croisée. En effet, $(a,b,c,d) = (2,7,8,9)$ est l'unique quadruplet tel que $a < b < c < d$ vérifiant $\texttt{Cl}(a) = \texttt{Cl}(c)$ et $\texttt{Cl}(b) = \texttt{Cl}(d)$ et ces quatre éléments sont bien dans la même classe.

En revanche, $\mathscr{R} = \{\{2,5\},\{4,7,9\},\{8\}\}$ n'est pas une partition non croisée. En effet, $\texttt{Cl}(2) = \texttt{Cl}(5)$ et $\texttt{Cl}(4) = \texttt{Cl}(7)$ mais $\texttt{Cl}(2) \neq \texttt{Cl}(4)$.

Le terme « non croisé » découle naturellement des représentations picturales des partitions (figure 1).

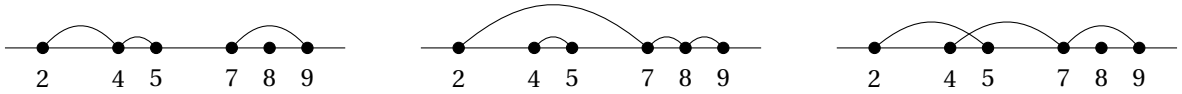


FIGURE 1 – Représentations picturales de $\mathscr{P}, \mathscr{Q}, \mathscr{R}$.

Dans la suite, on s'intéresse uniquement aux partitions d'un sous-ensemble fini de $\mathbb{N}$. On les représente en `OCaml` à l'aide de listes de listes croissantes d'entiers triées dans l'ordre lexicographique (c'est-à-dire l'ordre du dictionnaire).

Par exemple, les partitions $\mathscr{P} = \{\{2,5,4\},\{8\},\{7,9\}\}$ et $\mathscr{R} = \{\{9,4,7\},\{5,2\},\{8\}\}$ sont respectivement représentées par les listes `p=[[2;4;5];[7;9];[8]]` et `r=[[2;5];[4;7;9];[8]]`.

1.1 Exemples et fonctions sur les partitions

Question 1 : Justifier brièvement que $\mathscr{P} = \{\{1,7\},\{2\},\{3,4,5\},\{6\}\}$ est une partition non croisée de $[\![7]\!]$ et que $\mathscr{Q} = \{\{1,6\},\{2\},\{3,4,5,7\}\}$ n'en est pas une.

Question 2 : Parmi les ensembles suivants, indiquer sans justification lesquels sont des partitions de $[\![5]\!]$. Parmi les partitions, préciser sans justification lesquelles sont non croisées.

1. $\mathscr{P}_1 = \{\{1,3\},\{2,4,5\}\}$,
2. $\mathscr{P}_2 = \{\{1,3\},\{1,2\},\{4,5\}\}$,
3. $\mathscr{P}_3 = \{\{1,3\},\{2,4\}\}$,
4. $\mathscr{P}_4 = \{\{1,4,5\},\{2,3\}\}$.

Question 3 : Décrire l'ensemble des partitions non croisées de $[\![4]\!]$.

Dans la suite de cette section, et sans que ce soit rappelé à chaque question, un ensemble d'entiers sera toujours représenté en `OCaml` par une liste de type `int list` dont les données apparaissent dans un ordre strictement croissant sans qu'on n'ait jamais besoin de vérifier cette hypothèse. De même tous les entiers manipulés seront supposés positifs sans qu'on n'ait jamais besoin de le vérifier.

Question 4 : Écrire une fonction[1] `mem : int -> int list -> bool` telle que `mem x lst` s'évalue à `true` si et seulement si la valeur `x` apparaît dans la liste `lst`, en temps linéaire sur le minimum entre `x` et la longueur de `lst` (sans avoir à justifier cette complexité).

Question 5 : Écrire une fonction[2] `aucun_vide : int list list -> bool` telle que `aucun_vide lst` s'évalue à `true` si et seulement si aucune sous-liste de `lst` n'est vide, en temps linéaire sur la longueur de `lst` (sans avoir à justifier cette complexité).

Question 6 : On donne la fonction suivante :

```
1 (** inclus lst1 lst2 s'évalue à true si et seulement si l'ensemble
2     représenté par lst1 est inclus dans celui représenté par lst2 *)
3 let rec inclus (lst1 : int list) (lst2 : int list) =
4   match lst1 with
5   | [] -> true
6   | x :: xs -> mem x lst2 && inclus xs lst2
```

6a. Prouver la correction totale de la fonction `inclus`.

6b. Montrer que la complexité temporelle de la fonction `inclus` est en $\mathcal{O}(n_1 n_2)$, où n_1 est la longueur de `lst1` et n_2 celle de `lst2`.

6c. En tenant compte du fait que les ensembles sont supposés représentés par des listes dont les données apparaissent dans un ordre strictement croissant, proposer une autre implantation de `inclus` avec une meilleure complexité en fonction de n_1 et n_2. Donner et Justifier la complexité de cette nouvelle version.

1. Il est probable que le type de votre fonction tel que donné par le compilateur `OCaml` soit en fait `'a -> 'a list -> bool`, cela n'a pas d'importance. Il faut que cette fonction soit correcte si le type `'a` est `int`.
2. Idem avec le type `'a list list -> bool`.

Question 7 : On donne la fonction suivante :

```
(** depasse a p est vrai si et seulement si au moins une des
    sous-listes de p contient un element qui n'appartient pas à a *)
let rec depasse (a : int list) (p : int list list) : bool =
  match p with
  | [] -> false
  | x :: ps -> not (inclus x a) || depasse a ps
```

On admet la correction totale de cette fonction. Donner l'ordre de grandeur exact de sa complexité en fonction de ses paramètres, en supposant que c'est l'implantation de `inclus` donnée dans l'énoncé qui est utilisée. Justifier la réponse.

Question 8 : On donne la fonction suivante :

```
let rec occ x p =
  let rec nb x lst =
    match lst with
    | [] -> 0
    | y :: ys -> (if x = y then 1 else 0)  + nb x ys
  in match p with
     | [] -> 0
     | p1 :: ps -> nb x p1 + occ x ps
```

6a. Que fait la fonction auxiliaire `nb` ? Justifier la réponse.
6b. Exécuter à la main l'appel `occ 8 [[2;4;5;8];[7;8;9];[];[8]]`.
6c. Prouver la correction totale de la fonction `occ`.

Question 9 : Écrire une fonction[3] `couverture : int list -> int list list -> bool` telle que `couverture a p` s'évalue à `true` si et seulement si `p` contient tous les elements de `a` exactement une fois.

Question 10 : En utilisant les fonctions des questions précédentes, écrire une fonction[4]
`est_partition : int list -> int list list -> bool`
telle que `est_partition a p` s'évalue en `true` si et seulement si `p` est une partition de `a`.

1.2 Nombre de partitions non croisées

À partir de la représentation graphique d'une partition telle que donnée en figure 1, il est assez rapide de se convaincre que le nombre de partitions non croisées d'un sous-ensemble de $\mathbb{N}$ dépend uniquement de con cardinal et non de ses éléments. Pour tout $n \geq 1$, on note C_n le nombre de partitions non croisées d'un sous-ensemble de $\mathbb{N}$ de cardinal n. Par convention, on pose $C_0 = 1$.

> **Définition — Partition non croisée emboîtée.** Soit A un ensemble fini de $\mathbb{N}$ et $\mathscr{P}$ une partition non croisée de A. On dit que $\mathscr{P}$ est *emboîtée* si le minimum m et le maximum M de A appartiennent à la même classe : $\mathrm{Cl}(m) = \mathrm{Cl}(M)$.

Par exemple, $\mathscr{S} = \{\{1,2,5,9\}.\{3,4\},\{6,7,8\}\}$ est une partition non croisée et emboîtée de $[\![9]\!]$.

Le terme « emboîtée » découle naturellement des représentations picturales des partitions (figure 2).

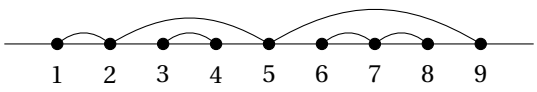


FIGURE 2 – Représentations picturales de $\mathscr{S}$.

3. Idem avec le type `'a list -> 'a list list -> bool`.
4. Idem avec le type `'a list -> 'a list list -> bool`.

À nouveau, il est assez rapide de se convaincre à partir de la représentation graphique d'une partition que le nombre de partitions non croisées emboîtées d'un sous-ensemble de $\mathbb{N}$ dépend uniquement de son cardinal et non de ses éléments. On note D_n le nombre de partitions non croisées emboîtées d'un sous-ensemble de $\mathbb{N}$ de cardinal n.

Question 11 : Montrer que pour tout n, on a $C_n = D_{n+1}$.

Question 12 : Si $A \subset \mathbb{N}$, on note NC(A) (resp. NCE(A)) l'ensemble des partitions non croisées (resp. non croisées emboîtées) de A, avec comme convention NC($\emptyset$) = NCE($\emptyset$) = $\{\emptyset\}$. On regardant le plus grand élément dans la même classe que 1, montrer que la fonction

$$\begin{array}{rccl} \Phi_n : & \bigcup_{i=1}^{n+1} \mathrm{NCE}([\![i]\!]) \times \mathrm{NC}([\![i+1, n+1]\!]) & \to & \mathrm{NC}([\![n+1]\!]) \\ & (Q_1, Q_2) & \mapsto & Q_1 \cup Q_2 \end{array}$$

est une bijection.

Question 13 : En admettant que deux ensembles finis en bijection ont même cardinal[5], montrer que

$$C_{n+1} = \sum_{k=0}^{n} C_k C_{n-k}\,.$$

Question 14 : Écrire une fonction `calcul_C : int -> int` qui prend en argument un entier n et calcule C_n.

2 Permutations de n polynômes

Le langage de programmation utilisé dans cette partie est le langage `C`.

Dans ce problème, on s'intéresse à des polynômes à coefficients entiers qui s'annulent en 0. Un tel polynôme P s'écrit $P(x) = a_1 x + a_2 x^2 + \cdots + x_m x^m$. Le but de ce problème est d'étudier la position relative autour de l'origine de plusieurs polynômes de ce type.

Dans toute la suite, le terme *polynôme* désigne un polynôme à coefficients entiers qui s'annule en 0.

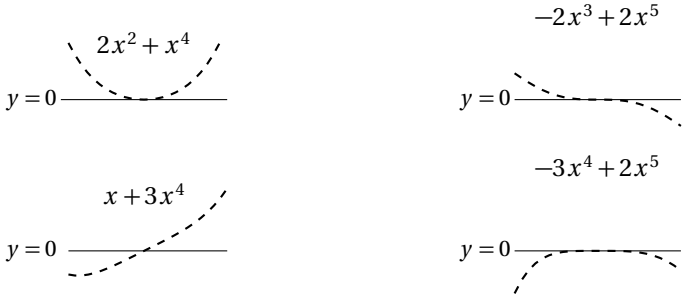
Dans tout le problème, les polynômes sont représentés par le type `int *`. Si le polynôme $P(x) = a_1 x + a_2 x^2 + \cdots + a_m x^m$ est représenté par

```
int *p;
```

alors `p[0]` aura pour valeur m et pour tout $i \in [\![1, m]\!]$, `p[i]` aura pour valeur a_i.

Question 15 : Afin de se familiariser avec cette représentation, écrire une fonction `exemple` qui prend en argument un entier m et crée un polynôme de degré m dont les coefficients des x^i avec i pair sont nuls et les coefficients des x_i avec i impair sont égaux à 1.

Nous commençons notre étude par quelques observations. Voici des exemples de graphes de polynômes autour de l'axe des abscisses.



5. Pas d'inquiétude, ce sera démontré en cours de mathématiques au second semestre !

On remarque que le comportement au voisinage de l'origine est décrit par le premier monôme $a_k x^k$ dont le coefficient a_k est non nul (les coefficients $a_1, \dots, a_{k-1}$ étant donc tous nuls). En effet, quand x est petit, le terme $a_{k+1}x^{k+1} + \cdots + a_m x^m$ est négligeable devant le terme $a_k x^k$. Cet entier k est la *valuation* du polynôme à l'origine. Par exemple, la valuation du polynôme $-2x^3 - 6x^5 + 4x^7$ est 3. On remarque alors les règles suivantes au voisinage de l'origine :

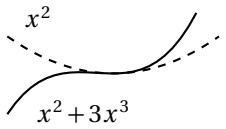
- Si la valuation k est *paire*, le graphe du polynôme reste du *même* côté de l'axe des abscisses.
- Si la valuation k est *impaire*, le graphe du polynôme *traverse* l'axe des abscisses.

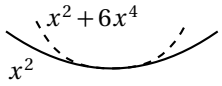
Question 16 : Écrire une fonction `valuation` qui prend en argument un polynôme P et renvoie sa valuation. Par convention, cette fonction renverra 0 si P est le polynôme nul.

Question 17 : Écrire une fonction `signe_neg` qui prend en argument un polynôme P et renvoie :

- un entier strictement négatif si $P(x) < 0$ pour x *négatif* assez petit ;
- 0 si P est le polynôme nul ;
- un entier strictement positif si $P(x) > 0$ pour x *négatif* assez petit ;

On s'intéresse maintenant aux positions relatives autour de l'origine des graphes de deux polynômes P_1 et P_2. La figure suivante montre les graphes de polynômes autour de l'origine.





On remarque que le comportement de ces graphes dépend de la parité de la valuation de la différence $P_1 - P_2$:

- Si la valuation de $P_1 - P_2$ est *paire*, les deux graphes se touchent mais ne se traversent pas à l'origine.
- Si la valuation de $P_1 - P_2$ est *impaire*, les deux graphes se traversent à l'origine.

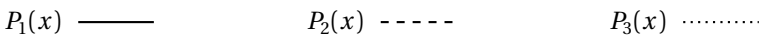
Question 18 : Écrire une fonction `difference` qui prend en arguments deux polynômes P_1 et P_2 et qui renvoie la différence des polynômes $P_1 - P_2$.

Question 19 : Écrire une fonction `compare_neg` qui prend en arguments deux polynômes P_1 et P_2 et qui renvoie :

- un entier strictement négatif si $P_1(x) < P_2(x)$ pour x *négatif* assez petit ;
- 0 si les deux polynômes P_1 et P_2 son égaux ;
- un entier strictement positif si $P_1(x) > P_2(x)$ pour x *négatif* assez petit ;

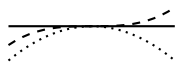
On admettra sans démonstration que la fonction `compare_neg` définit une relation d'ordre et on suppose qu'on dispose d'une fonction `compare_pos` similaire à la fonction `compare_neg` pour comparer deux polynômes pour x *positif* assez petit.

Enfin, passons à l'étude des graphes de trois polynômes. Les figures ci-après montrent les positions relatives de trois polynômes P_1, P_2 et P_3 autour de l'origine, avec la légende suivante :

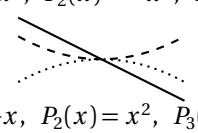


$P_1(x) = x^2,\ P_2(x) = 0,\ P_3(x) = -x^2$

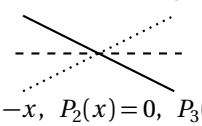
$P_1(x) = x^2,\ P_2(x) = -x^3,\ P_3(x) = 0$



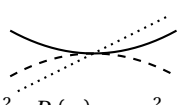
$P_1(x) = 0,\ P_2(x) = x^3,\ P_3(x) = -x^2$



$P_1(x) = -x,\ P_2(x) = x^2,\ P_3(x) = -x^2$



$P_1(x) = -x,\ P_2(x) = 0,\ P_3(x) = x$



$P_1(x) = x^2,\ P_2(x) = -x^2,\ P_3(x) = x$

Le choix de ces polynômes est fait pour qu'à chaque fois les inégalités $P_1(x) > P_2(x) > P_3(x)$ soient vérifiées pour x légèrement négatif. Maintenant, observons les positions relatives de ces graphes pour x légèrement positif. On remarque que l'ordre des courbes est *permuté* : on passe de l'ordre $P_1(x) > P_2(x) > P_3(x)$ à un autre ordre. La donnée des trois polynômes P_1, P_2 et P_3 définit donc une unique *permutation* π de $\{1,2,3\}$ telle que $P_{\pi(1)}(x) > P_{\pi(2)}(x) > P_{\pi(3)}(x)$, pour x positif et assez petit. On note que les *six* permutations de $\{1,2,3\}$ sont possibles, comme le montrent les six exemples ci-dessus.

De manière générale, on dit qu'une permutation π de $\{1,2,\dots,n\}$ *permute* les polynômes $P_1, P_2, \dots, P_n$ si et seulement si :

$$\begin{array}{lll} & P_1(x) > P_2(x) > \cdots > P_n(x) & \text{pour } x \textit{ négatif} \text{ assez petit} \\ \text{et} & P_{\pi(1)}(x) < P_{\pi(2)}(x) < \cdots < P_{\pi(n)}(x) & \text{pour } x \textit{ positif} \text{ assez petit.} \end{array}$$

Ce qui était vrai pour trois polynômes ne l'est plus à partir de quatre polynômes : il existe des permutations qui ne permutent aucun ensemble de polynômes $P_1, P_2, \dots, P_n$.

Dans la suite, toute permutation π de $\{1,2,\dots,n\}$ sera représentée par un tableau de taille $n+1$ dont la case d'indice 0 contient la valeur n et, pour tout $i \in [\![1,n]\!]$, la case d'indice i contient $\pi(i)$.

On considère le type suivant :

```
struct suite {
  int longueur;     // n+1 = taille du tableau p
  int **p;              // tableau de polynomes
};
```

qui permet de représenter une suite de polynômes : chaque case de l'attribut `p` d'indice strictement positif référence un polynôme de la suite. Le nombre de polynômes de la suite est donné par l'attribut n. La case d'indice 0 de l'attribut `p` n'est pas utilisée.

Question 20 : Écrire une fonction `tri` qui prend en argument une suite `s` de polynômes (sous la forme d'un `struct suite`) et la trie en utilisant la fonction `compare_neg` de telle sorte que l'on ait

```
s.p[1](x)>s.p[2](x)> ... >s.p[s.n](x)
```

pour x négatif et assez petit.

Question 21 : Écrire une fonction

```
/** pi est une permutation de {1,...,pi[0]}
* s est une suite de polynômes de longueur pi[0]
* et référence un tableau de polynômes qui a été trié par la fonction tri
*/
bool verifier_permut(int *pi, struct suite s);
```

qui teste si `pi` permute les polynômes `s.p[1]`, `s.p[2]`, ..., `s.p[s.n]`.