TD N°

2

Permutations, tris et polynômes

Nous travaillerons aujourd'hui sur les deux dernières épreuves de 2 heures de l'X/ENS Cachan. Comme il s'agit d'une épreuve écrite, on s'attachera à rédiger des solutions sur papier, présentées de manière claire, après avoir travaillé sur brouillon-papier et brouillon-écran.

1 Sujet 2011 - Permutations

On ne s'occupera aujourd'hui que des premières questions destinées à se familiariser avec ce genre d'épreuve et dégager quelques réflexes qui aideront à traiter le second sujet.
Si on rédige en Maple, on peut utiliser pour représenter les permutations (puis les polynômes), des listes ou des tableau (type `array`).
Les plus commodes à utiliser sont les listes. Nous introduisons ici les primitives `allouer` et `taille` souvent introduites dans les sujets même si la première est inutile avec les listes de Maple. Elles faciliteront cependant la tâche du correcteur.

```
> allouer := proc(m)
    RETURN([0 $ m])
  end:
> taille := proc(t)
    RETURN(nops(t))
  end:
```

1.1 Ordre d'une permutation

1. Diverses solutions sont possibles : réfléchir à ce qui empêche une *application* d'être une permutation. En Maple, les deux éléments de $\mathcal{B}_2$ sont `true` et `false`. Mais avant tout, comment représenter une permutation à l'aide d'une liste ?
2. Normalement, il ne faut pas oublier de tester si la composition est possible avant de l'effectuer mais l'énoncé nous dispense de le faire...
3. Si `u` est votre inverse, que vaut `u[t[i]]` ?
4. No comment...
5. On pourra introduire la liste correspondant à l'identité.

2 Sujet 2010 - Échangeurs de polynômes

Nous allons travailler sur des listes plutôt qu'avec des tableaux. On a souvent besoin de considérer la *tête* d'une liste qui est son premier élément et la *queue* d'une liste qui est la liste privée de son premier élément.
Nous pouvons donc introduire deux primitives :

```
> Tete := proc(L)
   RETURN(L[1])
  end:
> Queue := proc(L)
   RETURN(L[2..-1])
  end:
```

Pour la beauté du geste, vous essaierez d'éviter les boucles *pour* ou *tant que* pour répondre aux 5 premières questions au moins.

1. Il est plus léger ici d'utiliser l'algorithme de HÖRNER pour évaluer un polynôme. Prenons l'exemple de $P(x) = 3x^5 - 2x^4 + 7x^3 + 2x^2 + 5x - 3$. Le calcul classique nécessite 5 additions et 15 multiplications.

   On peut faire pas mal d'économies de calcul en suivant le schéma suivant :

$$\begin{aligned} P(x) &= \underbrace{a_n x^n + \cdots + a_2 x^2 + a_1 x}_{\text{on met } x \text{ en facteur}} + a_0 \\ &= \left(\underbrace{a_n x^{n-1} + \cdots + a_2 x}_{\text{on met } x \text{ en facteur}} + a_1 \right) x + a_0 \\ &= \cdots \\ &= (\cdots(((a_n x + a_{n-1})x + a_{n-2})x + a_{n-3})x + \cdots)x + a_0 \end{aligned}$$

Ici cela donne $P(x) = (((((3x)-2)x+7)x+2)x+5)x-3$ c'est-à-dire 5 multiplications et 5 additions. En fait il y a au maximum 2×degré de P opérations (voire moins avec les zéros).

Proposez ici une version récursive utilisant la liste des coefficients et nos primitives `Tete` et `Queue`.

2. Encore une fois, une version récursive est rapide à écrire. Vous aurez peut-être besoin de distinguer trois conditions. Utilisez alors la structure conditionnelle :

```
if ... then ...
elif ... then ...
else ...
fi
```

3. Une version récursive est attendue. Vous aurez sûrement besoin de concaténer deux listes ou d'ajouter un élément à une liste.

   Pour ajouter un 0 en bout de liste par exemple, on peut faire :

```
liste := [op(liste),0]
```

4. Une structure conditionnelle `if...elif...else...fi` traduit l'énoncé. On pourra utiliser la commande `signum(n)` qui renvoie 1 si $n > 0$, -1 si $n < 0$ et 0 sinon.

5. Voici une petite question qui cache un chapitre de l'option informatique consacrée aux tris !... Le sujet est vaste... Pour votre culture, nous aborderons tout de même deux tris possibles et leur complexité : le tri par insertion et le tri fusion.

   Tri par insertion C'est le tri qui correspond à un joueur qui classe ses cartes.

   Nous privilégierons une version récursive bien sûr.

   Commencez par créer une procédure récursive qui insère un élément dans une liste. Comme vous avez lu tout le sujet, vous savez qu'à la question suivante, il sera utile de trier une liste avec deux relations d'ordre différentes.

   Vous allez donc créer une procédure `insere := proc(Ordre,P,Liste)` qui insère le polynôme `P` dans la liste de polynômes `Liste` selon l'ordre croissant correspondant à `Ordre` (qui sera par exemple `Compare_neg` dans cette question).

   Pour obtenir la liste triée, il suffit de trier un à un les éléments de la liste avec une procédure récursive `tri := proc(Ordre,Liste)` qui utilise `insere`.

   Par exemple, vous devriez obtenir :

```
> tri(Compare_neg,[[1],[0,-1],[0,1]]);

                        [[0, 1], [0, -1], [1]]
```

   qui correspond à la sixième figure proposée dans le sujet.

   Combien d'opérations élémentaires la procédure va-t-elle effectuer dans le pire des cas ?

Tri fusion On utilise la méthode diviser pour régner : on « coupe » le tableau à trier en deux, on trie chaque tableau et on fusionne les deux tableaux obtenus, de manière récursive.

Il faut donc commencer par créer une procédure

```
divise := proc(Ordre,L)
```

qui divise la liste `L` en deux listes de longueurs égales ou presque, selon la parité de la taille.

Il faudra ensuite créer une procédure récursive

```
fusionne := proc(Ordre,L1,L2)
```

qui prend deux listes triées et en fait une seule liste triée.

On utilise ensuite ces deux procédures pour créer une procédure récursive

```
tri_f := proc(Ordre,Liste)
```

qui trie la liste `L`.

Pour la *complexité*, on peut supposer que la division s'effectue à temps constant et que la fusion de deux listes triées est de l'ordre de la taille n du tableau.

Soit K_n la complexité du tri d'une liste de taille n.

On obtient donc que K_n est de l'ordre de $2K_{\lfloor n/2\rfloor}+n$.

Il s'agit donc d'étudier la suite de terme général $u_n = 2u_{\lfloor n/2\rfloor}+n$ avec $u_0 = u_1 = 0$.

Montrez que u_n est croissante.

En introduisant la suite définie par $x_k = u_{2^k}$, montrez que :

$$n\lfloor\log_2(n)\rfloor \leqslant u_n \leqslant 2n\left(\lfloor\log_2(n)\rfloor+1\right)$$

6. Tout est expliqué... Après ce gros effort de récursion, vous pouvez utiliser une boucle *pour*. Utilisez à bon escient `RETURN` qui permet de sortir de la procédure, ce qui est adapté à ces procédure de vérification pour économiser du temps : on en sort dès qu'on tombe sur un os.

```
> verifier_permute([2,3,1],[[0,1],[0,-1],[1]]);

                                true
```

7. On peut utiliser ici une triple boucle *pour* indexée sur les coefficients a, b et c.

   Attention ! < est une relation d'ordre binaire en Maple ce qui interdit l'utilisation directe de `a<b<c<d`. On passera par :

   ```
   (x<y)and (y<z)
   ```

   Pour traduire la double inégalité $x < y < z$.

```
> est_echangeur_aux([2,4,1,3],2);

                                true

> est_echangeur_aux([2,4,1,3],1);

                               false
```

8. Pas de difficulté particulière :

```
> est_echangeur([2,4,1,3]);

                               false
```

9. L'utilisation de `sum` dans une récursion est peu fructueuse en Maple. On préférera une double boucle.

```
> nombre_echangeurs(4);

                                22
```

10. On traduit simplement l'énoncé.
11. C'est la question compliquée...

```
> enumerer_echangeurs(4);
[
[1,2,3,4],[1,2,4,3],[1,3,2,4],[1,3,4,2],[1,4,2,3],[1,4,3,2],[2,1,3,4],
[2,1,4,3],[2,3,1,4],[2,3,4,1],[2,4,3,1],[3,1,2,4],[3,2,1,4],[3,2,4,1],
[3,4,1,2],[3,4,2,1],[4,1,2,3],[4,1,3,2],[4,2,1,3],[4,2,3,1],[4,3,1,2],
[4,3,2,1]
]
```

3 Un petit oral de Centrale pour la route...

2 - 1 Centrale, PC 2010

1. Soit $n \in [\![0,10]\!]$. Montrer en utilisant Maple qu'il existe $P_n \in \mathbb{R}[X]$ tel que
$$\forall t \in \mathbb{R}, \quad \sin(nt) = P_n(\cos t)\sin t$$
2. Soit $n \in \mathbb{N}$. Montrer qu'il existe un unique $P_n \in \mathbb{R}[X]$ tel que
$$\forall t \in \mathbb{R}, \quad \sin(nt) = P_n(\cos t)\sin t$$
3. Montrer que $\forall n \in \mathbb{N}, \quad P_{n+2} = 2XP_{n+1} - P_n$
4. Écrire une procédure permettant de calculer P_n.

— Optionnel —

5. Montrer que l'application $(P,Q) \mapsto \displaystyle\int_{-1}^{1} P(t)Q(t)\sqrt{1-t^2}\,dt$ définit un produit scalaire sur $\mathbb{R}[X]$.
6. Montrer que la famille $(P_n)_{n\in\mathbb{N}}$ est une famille orthogonale de $\mathbb{R}[X]$.

Je vous donne la partie Maple. Vous noterez l'utilisation de `subs` pour les changements de variable, de l'option `remember` pour les tableaux récursifs, et `evalb(test)` qui évalue si un test est vrai ou faux.

```
# a)
for n from 0 to 10 do
    Q := expand(sin(n * t) / sin(t)):
    P[n] := subs(cos(t)=x, Q);
    od;

# c)
tchebychev := proc(n)
    option remember;
```

```
    if n=0 then
        RETURN 0
    elif n=1 then
        RETURN 1
    else
        RETURN expand(2*x*tchebychev(n-1) - tchebychev(n-2));
    fi;
    end:

for n from 0 to 10 do
    evalb(tchebychev(n) = P[n]);
    od;

tchebychev(35);
```

ÉCOLE POLYTECHNIQUE – ÉCOLE NORMALE SUPÉRIEURE DE CACHAN

ÉCOLE SUPÉRIEURE DE PHYSIQUE ET DE CHIMIE INDUSTRIELLES

CONCOURS D'ADMISSION 2011 — FILIÈRE MP HORS SPÉCIALITÉ INFO

FILIÈRE PC

COMPOSITION D'INFORMATIQUE – B – (XEC)

(Durée : 2 heures)

L'utilisation des calculatrices n'est pas autorisée pour cette épreuve.
Le langage de programmation choisi par le candidat doit être spécifié en tête de la copie.

⋆ ⋆ ⋆

Sur les permutations

La notion mathématique de permutation formalise la notion intuitive de réarrangement d'objets discernables. La permutation est une des notions fondamentales de la combinatoire, l'étude des dénombrements et des probabilités discrètes. Elle sert par exemple à étudier sudoku, Rubik's cube, etc. Plus généralement, on retrouve la notion de permutation au cœur de certaines théories des mathématiques, comme celle des groupes, des déterminants, de la symétrie, etc.

Une *permutation* est une bijection d'un ensemble E dans lui-même. On ordonne un ensemble E fini de taille n en numérotant ses éléments à partir de 1 : $x_1, x_2, \dots x_n$. En pratique, puisque seuls les réarrangements des éléments de E nous intéressent, on considère l'ensemble E_n des *indices* qui sont les entiers de 1 à n, bornes comprises. On représente alors simplement une application f sur E_n par un tableau t de taille n dont les éléments sont des indices. Autrement dit, $f(k)$ est $t[k]$, où $t[k]$ désigne le contenu de la case d'indice k du tableau t, et $t[k]$ est lui même un indice. On notera que f est une permutation, si et seulement si les contenus des cases de t sont exactement les entiers de E_n.

Dans tout le problème, les tableaux sont indicés à partir de 1. L'accès à la $i^{\text{ème}}$ case d'un tableau t de taille n est noté $t[i]$ dans l'énoncé, pour i entier compris entre 1 et n au sens large. Quel que soit le langage utilisé, on suppose que les tableaux peuvent être passés comme arguments des fonctions et renvoyés comme résultat. En outre, il existe une primitive allouer(n) pour créer un tableau de taille n (le contenu des cases du nouveau tableau ont des valeurs inconnues), et une primitive taille(t) qui renvoie la taille du tableau t. Enfin, les booléens vrai et faux sont utilisés dans certaines questions de ce problème. Bien évidemment, le candidat reste libre d'utiliser les notations propres au langage dans lequel il compose.

Partie I. Ordre d'une permutation

Question 1 Écrire une fonction estPermutation(t) qui prend une application (représentée par un tableau d'entiers t) en argument et vérifie que t représente bien une permutation. Autrement dit, la fonction renvoie vrai si t représente une permutation et faux sinon.

On suppose désormais, sans avoir à le vérifier, que les tableaux d'entiers (de taille n) donnés en arguments aux fonctions à écrire représentent bien des permutations (sur E_n). Dans cet esprit, on confond par la suite les permutations et les tableaux d'entiers qui les représentent en machine. Plus généralement, si l'énoncé contraint les arguments des fonctions à écrire, le code de ces fonction sera écrit en supposant que ces contraintes sont satisfaites.

Question 2 Écrire une fonction composer(t,u) qui prend deux permutations sur E_n en arguments et renvoie la composée $t \circ u$ de t et de u. On rappelle que la composée $f \circ g$ de deux applications est définie comme associant $f(g(x))$ à x.

Question 3 Écrire une fonction inverser(t) qui prend une permutation t en argument et renvoie la permutation inverse t^{-1}. On rappelle que l'application inverse f^{-1} d'une bijection est définie comme associant x à $f(x)$.

La notation t^k désigne t composée k fois, — la définition est correcte en raison de l'associativité de la composition. On définit l'*ordre* d'une permutation t comme le plus petit entier k non nul tel que t^k est l'identité.

Question 4 Donner un exemple de permutation d'ordre 1 et un exemple de permutation d'ordre n.

Question 5 En utilisant la fonction composer, écrire une fonction ordre(t) qui renvoie l'ordre de la permutation t.

Partie II. Manipuler les permutations

La *période* d'un indice i pour la permutation t est définie comme le plus petit entier k non nul tel que $t^k(i) = i$.

Question 6 Écrire une fonction periode(t,i) qui prend en arguments une permutation t et un indice i et qui renvoie la période de i pour t.

L'orbite de i pour la permutation t est l'ensemble des indices j tels qu'il existe k avec $t^k(i) = j$.

Question 7 Écrire une fonction estDansOrbite(t,i,j) qui prend en arguments une permutation t et deux indices, et qui renvoie vrai si j est dans l'orbite de i et faux sinon.

Une transposition est une permutation qui échange deux éléments *distincts* et laisse les autres inchangés.

Question 8 Écrire une fonction estTransposition(t) qui prend une permutation t en argument et renvoie vrai si t est une transposition et faux sinon.

Un cycle (simple) est une permutation dont exactement une des orbites est de taille strictement supérieure à un. Toutes les autres orbites, s'il y en a, sont réduites à des singletons.

Question 9 Écrire une fonction estCycle(t) qui prend une permutation t en argument et renvoie vrai si t est un cycle et faux sinon.

Partie III. Opérations efficaces sur les permutations

On commence par écrire une fonction qui calcule les périodes de tous les éléments de E_n et qui soit la plus efficace possible.

Question 10 Écrire une fonction periodes(t) qui renvoie le tableau p des périodes. C'est-à-dire que $p[i]$ est la période de l'indice i pour la permutation t. On impose un coût linéaire, c'est-à-dire que la fonction periodes effectue au plus $C \cdot n$ opérations avec C constant et n taille de t.

On envisage ensuite le calcul efficace de l'itérée t^k. On remarque en effet que $t^k(i)$ est égal à $t^r(i)$, où r est le reste de la division euclidienne de k par la période de i.

Question 11 Écrire une fonction itererEfficace(t,k) ($k \geq 0$) qui calcule l'itérée t^k en utilisant le tableau des périodes. On rappelle que t^0 est l'identité. Si besoin est, les candidats pourront utliser la primitive reste(a,b) qui renvoie le reste de la division euclidienne de a par b ($a \geq 0$, $b > 0$).

La fonction ordre de la question 5 n'est pas très efficace. En effet, elle procède à de l'ordre de o compositions de permutations, où o est l'ordre de la permutation passée en argument. Or, o est de l'ordre de $n^{\sqrt{n}}$ dans le cas le pire. On peut améliorer considérablement le calcul de l'ordre en constatant que l'ordre d'une permutation est le plus petit commun multiple des périodes des éléments.

Question 12 Donner un exemple de permutation dont l'ordre excède strictement la taille.

Question 13 Écrire une fonction pgcd(a,b) qui prend en arguments deux entiers strictement positifs a et b, et renvoie le plus grand diviseur commun de a et b. On impose un calcul efficace selon l'algorithme d'Euclide qui repose sur l'identité pgcd(a,b) = pgcd(b,r), avec r reste de la division euclidienne de a par b.

Question 14 Écrire une fonction ppcm(a,b) qui prend en arguments deux entiers strictement positifs a et b, et renvoie le plus petit commun multiple de a et b. On utilisera l'identité ppcm(a,b) = $(a \times b)$/pgdc(a,b).

ÉCOLE POLYTECHNIQUE

ÉCOLE SUPÉRIEURE DE PHYSIQUE ET CHIMIE INDUSTRIELLES

CONCOURS D'ADMISSION 2010

FILIÈRE MP - OPTION PHYSIQUE ET SCIENCES DE L'INGÉNIEUR **FILIÈRE PC**

COMPOSITION D'INFORMATIQUE

(Durée : 2 heures)

L'utilisation des calculatrices **n'est pas autorisée** pour cette épreuve.
Le langage de programmation choisi par le candidat doit être spécifié en tête de la copie.

⋆ ⋆ ⋆

Échangeurs de Polynômes

Dans ce problème on s'intéresse à des polynômes à coefficients réels qui s'annulent en 0. Un tel polynôme P s'écrit donc $P(x) = a_1x + a_2x^2 + \cdots + a_mx^m$. Le but de ce problème est d'étudier la position relative autour de l'origine de plusieurs polynômes de ce type.

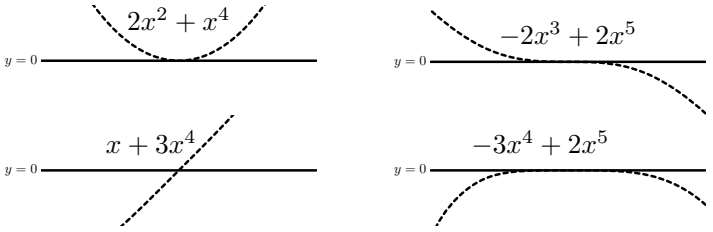
Dans tout le problème, les polynômes sont représentés par des tableaux de nombres flottants de la forme `[`a_1`;`a_2`;`$\cdots$`;`a_m`]`. Le nombre a_m peut être nul, par conséquent un polynôme donné admet plusieurs représentations sous forme de tableau. Ces tableaux sont indexés à partir de 1 et les éléments d'un tableau de taille m sont donc indexés de 1 à m. On suppose qu'il existe également une primitive `allouer(`m`)` pour créer un tableau de m cases. La taille m d'un tableau t est renvoyée par la primitive `taille(`t`)`. L'accès à la $i^{\text{ème}}$ case d'un tableau t est noté $t[i]$. Par ailleurs, on suppose que les tableaux peuvent être passés en argument ou renvoyés comme résultat de fonction, quel que soit le langage utilisé par le candidat pour composer. Enfin, les booléens `vrai` et `faux` sont utilisés dans certaines questions de ce problème. Le candidat est libre d'utiliser les notations propres à ces booléens dans le langage dans lequel il compose.

Le problème est découpé en deux parties qui peuvent être traitées de manière indépendante. Cependant, la **partie II** utilise les notions et notations introduites dans la **partie I**.

I. Permutation de n polynômes

Question 1 Afin de se familiariser avec cette représentation, écrire une fonction `evaluation` qui prend en arguments un polynôme P, représenté par un tableau, et un nombre flottant v, et qui renvoie la valeur de $P(v)$.

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

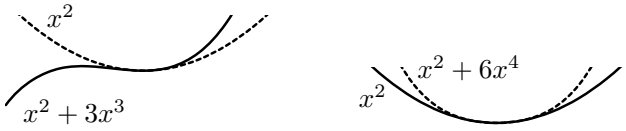


On remarque que le comportement au voisinage de l'origine est décrit par le premier monôme $a_k x^k$ dont le coefficient a_k est non nul (les coefficients $a_1, \ldots, a_{k-1}$ étant donc tous nuls). En effet, quand x est petit, le terme $a_{k+1}x^{k+1} + \ldots + a_m x^m$ est négligeable devant le terme $a_k x^k$. Cet entier k est la *valuation* du polynôme à l'origine. Par exemple, la valuation du polynôme $-2x^3 - 3x^5 + 4x^7$ est 3. On remarque alors les deux 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 2 Écrire une fonction `valuation` qui prend en argument un polynôme P et renvoie sa valuation. Par définition, cette fonction renverra 0 si P est le polynôme nul.

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 3 Écrire une fonction `difference` qui prend en arguments deux polynômes P_1 et P_2 (dont les tailles peuvent être différentes) et qui renvoie la différence des polynômes $P_1 - P_2$.

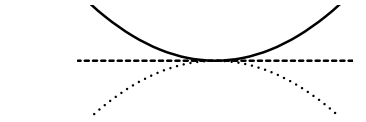
Question 4 Écrire une fonction `compare_neg` qui prend en arguments deux polynômes P_1 et P_2 et qui renvoie :

- un entier strictement négatif si $P_1(x)$ est plus petit que $P_2(x)$, pour x *négatif* assez petit
- 0 si les deux polynômes P_1 et P_2 sont égaux
- un entier strictement positif si $P_1(x)$ est plus grand que $P_2(x)$, pour x *négatif* assez petit.

On admettra sans démonstration que la fonction `compare_neg` définit une relation d'ordre.

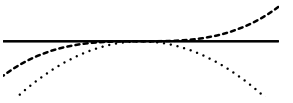
Enfin, passons à l'étude des graphes de trois polynômes. Les figures ci-après montrent les positions relatives de trois polynômes P_1, P_2 et P_3 autour de l'origine, avec la légende suivante :

$P_1(x)$ —— $P_2(x)$ ------- $P_3(x)$ ·····

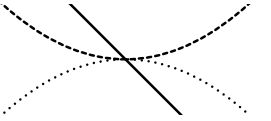


$P_1(x) = x^2 \quad P_2(x) = 0 \quad P_3(x) = -x^2$

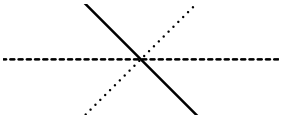
$P_1(x) = x^2 \quad P_2(x) = -x^3 \quad P_3(x) = 0$



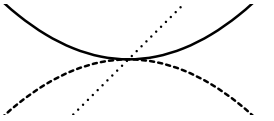
$P_1(x) = 0 \quad P_2(x) = x^3 \quad P_3(x) = -x^2$



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



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



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

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 $\{1, 2, \ldots, n\}$ *permute* les polynômes P_1, P_2, …, P_n si et seulement si :

$$P_1(x) > P_2(x) > \ldots > P_n(x) \qquad \text{pour } x \textit{ négatif} \text{ assez petit}$$
$$\text{et} \quad P_{\pi(1)}(x) > P_{\pi(2)}(x) > \ldots > P_{\pi(n)}(x) \quad \text{pour } x \textit{ positif} \text{ assez petit}$$

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, …, P_n.

Dans la suite, les permutations de $\{1, 2, \ldots, n\}$ seront représentées par des tableaux d'entiers de taille n, indexés à partir de 1 et contenant tous les entiers entre 1 et n.

Question 5 Écrire une fonction `tri` qui prend en argument un tableau t contenant n polynômes et qui le trie en utilisant la fonction `compare_neg`, de telle sorte que l'on ait $t[1](x) > t[2](x) > \cdots > t[n](x)$ pour x négatif et assez petit. Le candidat ne pourra pas utiliser pour cette question de fonction de tri prédéfinie dans la bibliothèque du langage qu'il utilise pour composer.

Question 6 Écrire une fonction `verifier_permute` qui prend en argument une permutation π de $\{1, 2, \ldots, n\}$ et un tableau t de même taille supposé trié par la fonction `tri`, et renvoie `vrai` si π permute les n polynômes $t[1]$, $t[2]$, …, $t[n]$ contenus dans t, et `faux` sinon. On pourra s'aider d'une fonction `compare_pos`, similaire à la fonction `compare_neg`, pour comparer deux polynômes pour x *positif* assez petit.

II. Échangeurs de n polynômes

Dans la suite, nous dirons qu'une permutation π de $\{1, 2, \ldots, n\}$ est un *échangeur* s'il existe n polynômes P_1, P_2, …, P_n tels que π permute ces polynômes. Nous allons maintenant écrire des fonctions qui répondent aux questions suivantes : Une permutation π est-elle un échangeur ? Peut-on dénombrer les échangeurs ? Peut-on énumérer les échangeurs ?

Une condition nécessaire et suffisante pour qu'une permutation soit un échangeur est la suivante : une permutation π de $\{1, 2, \ldots, n\}$ est un échangeur si et seulement si il n'existe aucun entiers a, b, c, d tels que $n \geq a > b > c > d \geq 1$ et

$$\pi(b) > \pi(d) > \pi(a) > \pi(c) \quad \text{ou} \quad \pi(c) > \pi(a) > \pi(d) > \pi(b) \tag{1}$$

Question 7 Écrire une fonction `est_echangeur_aux` qui prend en argument une permutation π de $\{1, 2, \ldots n\}$ et un entier d tel que $1 \leq d \leq n$ et qui renvoie `vrai` s'il n'existe aucun entier a, b et c tels que $n \geq a > b > c > d$ et vérifiant (1), et `faux` sinon.

Question 8 En utilisant la fonction `est_echangeur_aux`, écrire une fonction `est_echangeur` qui prend en argument une permutation π de $\{1, 2, \ldots, n\}$ et renvoie `vrai` si π est un échangeur, et `faux` sinon.

On admet sans démonstration que la relation de récurrence suivante permet de compter le nombre $a(n)$ de permutations de $\{1, 2, \ldots, n\}$ qui sont des échangeurs :

$$a(1) = 1, \qquad a(n) = a(n-1) + \sum_{i=1}^{n-1} a(i) \times a(n-i)$$

Question 9 Écrire une fonction `nombre_echangeurs` qui prend un entier n en argument et renvoie le nombre d'échangeurs $a(n)$. Enfin, les deux questions suivantes ont pour but d'énumérer tous les échangeurs de $\{1, 2, \ldots, n\}$.

Question 10 Écrire une fonction `decaler` qui prend en arguments un tableau t de taille n et un entier v, et renvoie un nouveau tableau u de taille $n+1$ tel que

$$\begin{cases} u[1] = v \\ u[i] = t[i-1] & \text{si } t[i-1] < v \text{ et } 2 \leq i \leq n+1 \\ u[i] = 1 + t[i-1] & \text{si } t[i-1] \geq v \text{ et } 2 \leq i \leq n+1 \end{cases}$$

L'algorithme que nous allons utiliser pour énumérer les échangeurs de $\{1, 2, \ldots, n\}$ consiste à énumérer successivement les échangeurs de $\{1, 2, \ldots, k\}$, pour tout k de 1 à n, dans un tableau t de taille $a(n)$. Si on suppose qu'un tableau t contient les m échangeurs de $\{1, \ldots, k\}$ entre les cases $t[1]$ et $t[m]$, on peut en déduire les échangeurs de $\{1, \ldots, k+1\}$ de la manière suivante : pour tout entier v entre 1 et $k+1$ et tout entier i entre 1 et m, on décale (à l'aide de la fonction `decaler`) l'échangeur $t[i]$ avec v puis on teste si le résultat est un échangeur (avec la fonction `est_echangeur_aux`).

Question 11 Écrire une fonction `enumerer_echangeurs` qui prend un entier n en argument et renvoie un tableau contenant les $a(n)$ échangeurs de $\{1, 2, \ldots, \mathtt{n}\}$. On pourra utiliser un second tableau pour stocker temporairement les nouveaux échangeurs.

* *
*