X Informatique MP/PC 2011 — Corrigé

Ce corrigé est proposé par Arnaud Borde (École polytechnique) ; il a été relu par Céline Chevalier (Enseignant-chercheur à l'université) et Guillaume Batog (ENS Cachan).

Cette épreuve est commune aux filières PC et MP option sciences de l'ingénieur. Elle ne compte que pour l'admission.

Le sujet propose l'étude des permutations de $\mathrm{E}_n = [\![1 ; n]\!]$ et de quelques-unes de leurs propriétés : ordre, période et orbite. Il est divisé en trois parties de tailles et difficultés équivalentes, à l'exception des questions 9 et 10 qui demandent davantage de réflexion.

- La première partie s'attache à définir les permutations, les composer, les inverser, et enfin à définir l'ordre d'une permutation.
- La deuxième introduit les notions de période et d'orbite. Elle exploite les cas particuliers importants que sont les transpositions et les cycles.
- La troisième partie utilise les propriétés mathématiques des permutations pour construire un algorithme efficace de calcul de l'ordre d'une permutation.

La plupart des questions demandent d'écrire du code. Dans ce corrigé, ce dernier a été rédigé en Maple car il s'agit du langage le plus utilisé en prépa, même s'il est plus destiné au calcul formel qu'à la programmation. D'autres langages, comme C++, Caml, Pascal, Java et Python, sont autorisés pour cette épreuve. Les fonctions demandées n'utilisant que la syntaxe de base de Maple, vous pourrez sans difficulté les réécrire dans le langage que vous préférez.

C'est un sujet plus austère que les années précédentes, mais il reste à la portée d'un élève ayant un peu préparé cette épreuve, qui est spécifique à l'X. L'énoncé dans son ensemble est clair et les questions sont toutes bien détaillées. Comme souvent, il était judicieux de lire l'énoncé dans son ensemble afin de repérer les questions qui pouvaient rapporter facilement des points.

Rédiger du code sur papier demande un soin particulier. Il est rare de produire une procédure juste dès le premier jet : un brouillon s'impose donc pour éviter de multiples ratures et rajouts. De même, afin de faciliter le travail du correcteur, il est important de l'indenter et de l'espacer afin d'en dégager la structure (quelles sont les parties qui le composent, à quelles boucles appartiennent les instructions). Il ne faut pas non plus hésiter à insérer des commentaires (avec # sous Maple) et à expliquer les grandes lignes de sa démarche.

Indications

I. Ordre d'une permutation

1 Chercher les deux conditions sous lesquelles un tableau ne représente pas une permutation.

2 Retranscrire dans le code la formule de l'énoncé pour chaque indice du tableau.

3 Par définition, $t^{-1}[t[i]] = i$ pour tout $i \in \mathrm{E}_n$.

4 Penser aux permutations circulaires.

5 Écrire une fonction auxiliaire qui teste si une permutation est l'identité.

II. Manipuler les permutations

6 Calculer successivement $t^k[i]$ pour $k = 1, 2, \ldots$ sans utiliser la fonction `composer`.

7 S'inspirer du code de la fonction `periode` pour parcourir l'orbite.

8 Une transposition est une permutation qui diffère de l'identité en exactement deux indices.

9 Vérifier que tous les indices i tel que $t[i] \neq i$ appartiennent bien à la même orbite.

III. Opérations efficaces sur les permutations

10 Pour obtenir un coût linéaire, remarquer que la somme des tailles des orbites vaut la taille de la permutation.

11 Traiter chaque élément de t séparément.

12 Composer par exemple deux permutations de taille 5 et d'ordres respectifs 2 et 3.

15 Utiliser $\texttt{ppcm}(a, b, c) = \texttt{ppcm}(a, \texttt{ppcm}(b, c))$.

Afin de rester proche du sujet tout en ayant des codes fonctionnels sous Maple, le code des deux primitives proposées par l'énoncé (`allouer` et `taille`) est donné ici. Ces deux fonctions font appel aux fonctions `Array` et `ArrayNumElems` de Maple que des élèves de CPGE ne sont pas censés connaître.

```
allouer := proc(n)
  RETURN( Array(1..n) );
end;

taille := proc(t)
  RETURN(ArrayNumElems(t));
end;
```

Dans ce cas précis, la fonction `allouer(n)` renvoie un tableau de taille `n` dont tous les éléments valent 0. Néanmoins, l'énoncé est ici strictement respecté et on suppose dans la suite que le contenu des cases après cette allocation est inconnu. De manière plus générale, il est important de veiller à l'initialisation des variables.

Il est aussi possible d'utiliser la structure de liste de Maple qui donnerait :

```
allouer := proc(n)
  RETURN( [seq(0, i = 1 .. n)] );
end;

taille := proc(t)
  RETURN(nops(t));
end;
```

Ici c'est la commande `seq` qui permet d'initialiser la liste avec le nombre d'éléments voulus (des zéros en l'occurence). Et c'est la fonction `nops` qui donne le nombre d'éléments dans la liste.

Dans Maple, contrairement aux autres langages de programmation, les indices de tableaux commencent par défaut à 1 et non à 0. Si vous utilisez un autre langage, il est préférable de préciser la convention que vous adoptez pour les indices car le sujet suppose qu'ils débutent à 1. Les rapports du concours des années précédentes recommandent d'ailleurs de suivre l'énoncé plutôt que les conventions du langage utilisé, quitte à écrire un code qui n'est pas compilable/exécutable. L'utilisation de la structure `Array` permet de choisir les indices de début et de fin du tableau.

Dans l'ensemble du corrigé, les conventions de Maple pour les variables booléennes sont utilisées : `true` pour `vrai` et `false` pour `faux`.

I. Ordre d'une permutation

1 Si `t` est une permutation, alors par définition (c'est une bijection) on y trouve une et une seule fois chaque entier de $[\![1 ; n]\!]$. Pour tout indice de t, effectuons un test en deux parties :

- une première où l'on regarde si l'entier appartient bien à E_n,
- une seconde où l'on vérifie que l'entier n'est pas déjà apparu.

Dans le code, cela représente une boucle `for` dans laquelle les deux tests sont effectués. Pour vérifier que chaque entier apparaît une seule fois, utilisons un tableau de valeurs booléennes `present` de la même taille que t et initialisé à `false`. Chaque élément `present[i]` permet de savoir si l'entier `i` a déjà été rencontré. Si l'on rencontre `i` alors que `present[i]` vaut `true`, alors `t` n'est pas une permutation et le code renvoie `false` ; sinon `present[t[i]]` est mis à jour à `true`. Si on sort de la boucle `for` sans retourner `false`, alors le tableau t est constitué de n entiers deux à deux distincts appartenant à E_n, de cardinal n. Ainsi, t représente bien une permutation de E_n et le code renvoie `true`.

```
estPermutation := proc(t)
  local compte, tailleT, i;
  tailleT := taille(t);
  present := allouer(tailleT);
  for i from 1 to tailleT do
    present[i] := false;
  od;

  # vérifie qu'on ne dépasse pas les bornes
  # et que de chaque entier est présent au plus une fois
  for i from 1 to tailleT do
    if t[i] < 1 or t[i] > tailleT then
      RETURN(false);
    elif present[t[i]] = true then
      RETURN(false);
    else
      present[t[i]] := true
    fi;
  od;

  RETURN(true);
end;
```

> C'est ici la fonction `RETURN` qui permet d'arrêter la procédure et de renvoyer la valeur `false`. Par défaut Maple renvoie la dernière valeur calculée, néanmoins il est plus prudent d'expliciter la valeur retournée avec un `RETURN`.
>
> L'énoncé simplifie grandement la suite en indiquant que les arguments satisfont les contraintes imposées. Ainsi, il est inutile d'alourdir le code avec quantité de tests vérifiant par exemple que les tableaux donnés en arguments représentent bien des permutations ou qu'ils ont la même taille lorsqu'on va vouloir les comparer ou les composer.

2 Appliquons la formule de l'énoncé mot pour mot : il s'agit de calculer $t[u[i]]$ pour chaque i. On réalise cela grâce à une boucle `for` qui permet de parcourir tous les entiers de E_n. Les résultats sont stockés dans un tableau `composee` qui est retourné à la fin de la procédure.

```
composer := proc(t, u)
  local composee, tailleT, i;
  tailleT := taille(t);
  composee := allouer(tailleT);
```