Maple TD 7 **Janvier 2007**

Ce TD a pour but de vous faire travailler un peu de programmation. Nous reverrons tout d'abord les notions de base à travers quelques exercices simples. Puis nous travaillerons sur les listes et nous aborderons les problèmes de tri (un grand classique). Enfin la dernière partie (pour les gens rapides ou motivés) est plus une ouverture. Ce TD est probablement un peu long, il me servira aussi à évaluer votre niveau. Ça serait bien que vous arriviez à terminer l'exercice 5.

1 Rappels de programmation

Quelques petits rappels, ca ne devrait pas vous poser de difficultés. N'hésitez pas à revoir vos anciens TDs et à utiliser l'aide de Maple.

Exercice 1

1. *Dans la syntaxe Maple pour les procédures, par exemple*

   ```
   [> maprocedure := proc(n::integer) option remember; ... end;
   ```

   À quoi sert l'instruction `integer` *? Comment aurait-on pu écrire ? Que dois-je écrire si la procédure prend non pas un entier mais un flottant ? une liste ? deux entiers ? un entier et une liste ?*
2. *À quoi sert l'instruction* `option remember` *? est-elle nécessaire ? discutez ses avantages et limites.*
3. *Écrire une procédure récursive* `Factorielle(n)` *retournant* $n!$ *(Rappel : une fonction est récursive quand elle s'appelle elle-même).*
4. *Écrire* `Factorielle(n)` *en style impératif (itératif) cette fois.*

2 Listes

On appelle *structures de données* les objets informatiques ayant pour but de modéliser des collections. Les **listes** et les **tableaux** sont les deux structures de données fondamentales. La différence majeure : les listes ont une *structure dynamique* (leur taille peut évoluer) tandis que celle des tableaux est *statique* (taille fixée à la définition). Nous allons travailler ici sur les listes.

Syntaxiquement, en Maple, une liste est une suite entre crochets d'éléments séparés par des virgules. Plus formellement, la structure de liste est *récursive*, et est définie de la manière suivante : une liste L est
- soit la liste vide : `[]`
- soit la concaténation d'une liste L' et d'un élément a : `L :=[a,op(L')]`

2.1 Opérations élémentaires

Les exemples qui suivent sont fondamentaux et doivent être parfaitement maîtrisés :

Définition d'une liste : `[> L := [1,4,9,16];`
Liste vide : `[> L := [];`
Longueur d'une liste : `[> nops(L);`
i-ème élément de la liste : `[> L[i];` i est un entier compris entre 1 et longueur(L)
Extraction d'une sous liste : `[> L[i..j];` $1 \leq i \leq j \leq$ longueur(L)
Ajout d'un élément en fin de liste : `[> L :=[op(L),23];`
Ajout d'un élément en début de liste : `[> L :=[23, op(L)];`
Définition "automatique" : `[> L :=[seq(k,k=1..5)];`
Opération sur tous les éléments : `[> map(x->x^2,L);`

Il est essentiel de noter le lien entre listes et séquences (idem, sans les crochets autour). Ainsi on peut construire des listes "complexes" en utilisant le constructeur de séquences `seq`. De plus, les transformations de liste en séquence (`op`) et réciproquement (`[.]`) sont très simples.
Testez un peu les outils ci-dessus, afin de vous assurer que vous savez tous les utiliser. N'hésitez pas à aller regarder dans l'aide. Pour afficher les éléments d'une liste L les un après les autres, vous pouvez faire : `[> for i in L do print(i) od;`

Exercice 2

1. *Définissez la suite* $[1, 4, 9, 16]$ *de trois manières différentes (méthode directe,* `map`*,* `seq`*).*
2. *Connaissez-vous la fonction Maple qui teste la parité d'un entier ? Regardez l'aide sur les mots* `even` *(pair en anglais) ou* `odd` *(impair en anglais). Définissez* P*, liste des entiers pairs de 0 à 12 (vous procéderez deux fois : une avec* `map`*, l'autre avec un test de parité). Supprimez le 3-ème élément de* P *(indication : regardez l'aide sur le mot* `list` *pour trouver les opérations sur les listes).*
3. *Connaissez-vous la fonction Maple qui teste la primalité d'un entier ? Regardez l'aide sur* `isprime`*. Essayez de comprendre le texte en anglais. La valeur que retourne* `isprime` *est-elle sûre ? Afficher les* 40 *premiers nombres premiers (en utilisant une boucle* ***while****,* `isprime` *et en stockant les valeurs dans une liste). Finalement la procédure* `isprime` *vous semble-t-elle sûre ?*

2.2 procédures utilisant des listes

Les exercices suivant ont pour objectif de vous entraîner à utiliser des listes dans des boucles, et de vous faire faire un peu de programmation. Il existe en général des fonctions Maple obtenant les mêmes résultats que les boucles suivantes.

> Faites attention lorsque vous modifiez une liste dans une boucle ! Une commande du style : `for i to nops(L) do ... od;` sera comprise par Maple comme : pour i allant de 1 à la taille que fait L avant de commencer la boucle faire ... Il est en général bien mieux de créer une nouvelle liste qui pourra changer de taille dans la boucle mais dont ne dépend aucune condition d'arrêt.

Exercice 3 *Sortez un crayon, un papier et lâchez votre clavier, pour cet exercice on va commencer par réfléchir un peu ! ! L'algorithme suivant est écrit en pseudo-code : c'est à dire un langage informel ressemblant à un langage de programmation (mais qui n'est pas du Maple strict ! !). Que fait cet algorithme ? Vous devez pouvoir exprimer simplement ce que vaut le résultat* m *par rapport à l'entrée* L*, et argumenter un minimum votre réponse. Si vous ne voyez pas trop, vous pouvez exécuter l'algorithme à la main sur de petits exemples, par exemple avec* $L = [2, 5, 3]$ *puis* $L = [5, 2, 4]$*. Une fois que vous m'aurez expliqué comment et pourquoi l'algorithme fonctionne, vous pourrez le coder en Maple.*

```
Entrée : un argument L de type list
Variables locales : m, size, i
size := longueur(L)
m := L[1]
Pour i entre 1 et size faire
      si m<L[i] alors m := L[i] fin si
fin boucle
Renvoyer m
Fin
```

Exercice 4

1. *Écrivez une procédure qui dit si oui ou non un élément* a *est dans une liste* L*, et à quelle place (s'il y a plusieurs fois le même élément, vous pouvez renvoyer la place du premier, ou bien la liste des places de toutes les occurrences de* a*).*
2. *Écrivez une procédure qui, à partir d'une liste* L *contenant des entiers naturels, renvoie une nouvelle liste contenant seulement les éléments non nuls de* L *(et dans le même ordre, bien sûr).*
3. *Écrivez une procédure qui inverse l'ordre des éléments d'une liste.*

3 Algorithmes de Tri

Le but de cette section est de vous faire programmer des procédures un peu plus compliquées. Nous utiliserons pour cela les algorithmes de tri, un grand classique. Un algorithme de tri prend en entrée une liste d'entiers L et renvoient une nouvelle liste contenant les mêmes éléments triés par ordre croissant. Les algorithmes proposés sont ordonnés par difficulté croissante.

Exercice 5 (Sélection) *Ce tri est basé sur une idée simple : il est facile de déterminer la position du plus petit élément de L, et sa nouvelle position dans la liste triée est connue puisqu'il doit être en première position ! Nous pouvons donc* **échanger** *ces deux éléments (le premier et celui contenant le minimum de la liste). Il suffit ensuite d'itérer cette idée avec la suite de la liste. Ainsi, après i itérations, la sous liste $L[1..i]$ contient les i plus petits éléments de L rangés par ordre croissant, et la sous-liste $L[i+1..n]$ les éléments restant à trier.*

1. *Commencez par faire quelques exemples à la main pour bien comprendre l'algorithme.*
2. *Écrivez une procédure* `Indice-min(L)` *qui étant donnée une liste d'entiers retourne l'indice du plus petit élément de cette liste.*
3. *Écrivez une procédure* `Echange(L,i,j)` *qui étant donnés une liste* `L` *et deux positions* `i` *et* `j` *dans cette liste, renvoie une nouvelle liste définie de la manière naturelle.*
4. *Écrivez enfin une procédure* `Selection(L)` *qui étant donnée une liste renvoie la liste triée associée obtenue par le tri par sélection.*

Exercice 6 (Insertion) *Ce tri est censé être le tri naturel pour trier un jeu de cartes. Le principe du tri est le suivant : On va ajouter successivement à leur place dans un tableau initialement vide tous les éléments de L. Il est en effet facile d'insérer à sa place un élément dans une liste déjà triée : il suffit d'avancer dans cette liste jusqu'à trouver un élément plus grand que l'élément à insérer.*

1. *Commencez par faire quelques exemples à la main pour bien comprendre l'algorithme.*
2. *Écrivez une procédure* `Inserer(L,a)` *qui étant donnée une liste d'entiers* `L` *supposée triée et un entier* `a` *retourne la liste obtenue en insérant* `a` *à sa place dans* `L`.
3. *Écrivez enfin une procédure* `Insertion(L)` *qui étant donnée une liste renvoie la liste triée associée obtenue par le tri par insertion.*

Exercice 7 (Fusion) *La particularité de ce tri est d'utiliser une méthode* ***récursive****. En effet, on découpe la liste en deux sous-listes, puis on trie récursivement les deux sous-listes. Ensuite vient une phase de recomposition, appelée* **fusion***. Celle-ci consiste, étant données deux listes supposées triées, à fusionner ces deux listes en une liste triée. Écrivez la procédure* `Fusion(L1,L2)` *définie ci-dessus, puis une procédure* `Tri-fusion(L)` *qui étant donnée une liste renvoie la liste triée associée obtenue par le tri par fusion.*

Exercice 8 (Quick Sort) *À nouveau, ce tri est récursif. Il consiste à choisir un élément arbitraire du tableau, appelé* pivot*. Il s'agit ensuite de réorganiser la liste L de sorte que tous les éléments inférieurs (resp. supérieurs) au pivot soient à gauche (resp. à droite) du pivot. Ainsi, le pivot est à sa place dans la future liste triée. On réapplique ensuite l'algorithme aux deux sous-listes des éléments inférieurs et supérieurs au pivot. Pour fixer les idées, on dira que le pivot est le premier élément de la liste. Écrivez les procédures* `Organise(L,a)` *et* `QuickSort(L)` *comme définies ci-dessus.*

4 Ouverture : Complexité algorithmique

Cette partie n'est nullement exigible aux concours. Voyez la plutôt comme une ouverture vers l'aspect plus théorique de l'informatique.

La notion de complexité algorithmique tente de répondre à des questions comme

- Combien coûte un algorithme (en temps de calcul ou en mémoire) ?
- De deux algorithmes donnés pour résoudre un même problème, lequel est le plus performant ?
- Si j'ai déjà un algorithme, est-il optimal (puis-je en trouver un meilleur) ?

Nous ne donnerons pas de définition formelle ici, et nous regarderons la complexité en temps. Intuitivement, nous dirons qu'un algorithme a pour complexité la fonction $f(n)$ si, étant donnée une entrée de taille n (ici une liste de longueur n), le nombre d'opérations que l'algorithme doit faire pour retourner le résultat est dans la classe $O(f(n))$. Les seules opérations que nous compterons ici seront les lectures des éléments de la liste. Exemple :

```
Entreé : L
Variables locales : m, l, i
l := longueur(L)
m := L[1]
Pour i entre 2 et l faire
      si m<L[i] alors m := L[i] fin si
fin boucle
Renvoyer m
Fin
```

Pour cet algorithme (que fait-il au fait ?), la complexité est $f(n) = 2n$ *dans le pire des cas*, et $f(n) = n$ *dans le meilleur des cas*. Comme c'est en fait la classe de complexité qui nous intéresse, nous retiendrons simplement que cet algorithme est de complexité $O(n)$ (pire et meilleur des cas) et dirons que c'est un algorithme linéaire[1].

Exercice 9 *Reprenez l'exemple ci-dessus. Justifiez les résultats de complexité de $f(n)$.*

Les algorithmes naïfs de tri sont en $O(n^2)$ (pire des cas). Le Quick Sort est en $O(n \log n)$ (cas moyen). Il est en fait possible de montrer que cette borne de complexité est optimale, mais c'est plus dur...

Exercice 10 *Reprenez les différents algorithmes de tri et*

1. *(*) Evaluez la complexité de ces algorithme dans le pire cas et dans le meilleur cas. Décrivez les cas où sont atteints ces valeurs de complexité.*
2. *(**) Sauriez-vous le* démontrer *?*

[1] Les algorithmes polynomiaux sont considérés comme utilisables "en vrai", au-delà (exponentiel par exemple), les temps de calcul deviennent démesurés.