

L1 Informatique

Algorithmique 2.2 – Exercices

1 Listes chaînées

1.1 Listes d'entiers

1. Définir un type liste permettant de manipuler des listes chaînées d'entiers, ainsi que les primitives suivantes : **initialiser**, **ajout_debut**, **supprime_premier**, **vide**, **affiche**, **appartient**, **longueur**.
2. Écrire les sous-programmes supplémentaires suivants, de manière récursive :
 - (a) Afficher les éléments d'une liste **L**, du dernier vers le premier ;
 - (b) Ajouter un entier **n** à la fin d'une liste **L** ;
 - (c) Ajouter un entier **n** dans une liste **L** après l'élément d'adresse **adr** donnée, ou en fin de liste si **adr** n'est pas dans **L** ;
 - (d) Retourner l'adresse de l'élément contenant la première occurrence d'un entier **n** dans une liste **L**, ou **nullptr** si **n** n'est pas dans **L** ;
 - (e) Ajouter dans une liste **L** un entier **n** après la première occurrence d'un entier **x** donné, ou en fin de liste si **x** n'est pas dans **L** ;
 - (f) Supprimer la première occurrence d'un entier **n** dans une liste **L** ;
 - (g) Compter le nombre d'occurrences d'un entier **n** dans une liste **L** ;
 - (h) Afficher les éléments de rang impair d'une liste **L** (le premier, le troisième, le cinquième, ...) ;
 - (i) Vérifier si une liste d'entiers **L** représente une suite de valeurs croissante (au sens large) ;
 - (j) Vérifier si une liste d'entiers **L** représente une suite de valeurs strictement monotone (croissante ou décroissante, au sens strict) ;
 - (k) Vérifier si une liste **L** est sans doublons ;
 - (l) Vérifier si une liste **L1** est incluse dans une liste **L2**, c'est-à-dire que tous les éléments de **L1** apparaissent, **dans le même ordre** mais pas nécessairement à la suite, dans **L2**. Par exemple, (1, 8, 4, 7) est incluse dans (3, 1, 8, 2, 4, 7, 9) mais pas dans (4, 3, 1, 5, 8, 7) ni dans (1, 2, 4, 7, 3).
3. Dérécursiver les primitives d'affichage et d'ajout en fin.
4. Écrire deux sous-programmes permettant de convertir une liste en tableau, et réciproquement.

1.2 Répertoire [TP]

On veut écrire un programme permettant de manipuler un répertoire de personnes en mémorisant, pour chacune des personnes, le nom, le prénom et le numéro de téléphone. Nous représenterons en mémoire un répertoire par une liste chaînée, et pour cela, on pourra utiliser les structures de données suivantes :

```
struct personne
{
    std::string nom;
    std::string prenom;
    std::string tel;
};

struct maillon
{
    personne val;
    maillon * suiv;
};

using repertoire = maillon *;
```

Écrire les sous-programmes suivants :

1. **void initialiserRepertoire (repertoire & R)**
2. **void ajouterEnTete (std::string nom, std::string prenom, std::string tel, repertoire & R)**
3. **void ajouterEnQueue (std::string nom, std::string prenom, std::string tel, repertoire & R)**
4. **void afficherPersonne (personne P)** : affichage des informations d'une personne sous la forme [nom, prénom, téléphone]
5. **void afficherRepertoire (repertoire R)**
6. **std::string telephone (std::string nom, std::string prenom, repertoire R)** : recherche la première occurrence d'une personne selon son nom et son prénom, puis retourne son numéro de téléphone (ou la chaîne vide si cette personne n'est pas dans le répertoire)
7. **int rechercherPosition (std::string nom, std::string prenom, repertoire R)** : recherche la première occurrence d'une personne selon son nom et son prénom, puis retourne sa position dans le répertoire (la 1ère personne a pour position 1 ; 0 sera retourné si cette personne n'est pas dans le répertoire).
8. **void ajouter (int position, std::string nom, std::string prenom, std::string tel, repertoire & R)** : ajoute une personne à une position donnée dans le répertoire (si la position est strictement supérieure à taille du répertoire, alors la personne est ajoutée en fin de répertoire).
9. **void supprimer (int position, repertoire & R)** : supprime la personne à la position donnée (si la position est strictement supérieure à taille du répertoire, alors on ne fait rien).
10. **void supprimer (std::string nom, repertoire & R)** : supprime du répertoire toutes les personnes portant ce nom.

1.3 Crible d'Eratosthène [TP]

Écrire un programme permettant d'afficher la liste des nombres premiers inférieurs ou égaux à un entier **n** donné par l'utilisateur. La méthode devra consister à construire une liste chaînée d'entiers éligibles (ordonnés du plus petit au plus grand), puis de supprimer, pour chaque élément de la liste, tous leurs multiples stricts.

1.4 Polynômes [TD/TP]

Cet exercice consiste à manipuler des polynômes (calcul de dérivées, somme et produit de polynômes). On représente un monôme M (de la forme $a \cdot x^b$) par un enregistrement composé d'un coefficient a et d'un degré b . Un polynôme P sera vu comme une somme de monômes ($P = M_1 + \dots + M_n$) et représenté comme une liste chaînées d'éléments de type monôme. Les monômes n'auront pas nécessairement à être ordonnés par degré.

1. Définir les structures de données **monome** et **polynome**.
2. Écrire une procédure **saisie (polynome & P, int N)** permettant de saisir un polynôme composé de **N** monômes → voir exemple en fin d'énoncé.
3. Écrire une procédure **affiche (polynome P)** permettant d'afficher un polynôme d'une manière similaire à celle de l'exemple. Ce sous-programme appellera une procédure **affiche (monome M)** à écrire au préalable.
4. Écrire une procédure **supprime (polynome P)** qui libère la mémoire allouée dynamiquement pour un polynôme.
5. Écrire une fonction entière **degre (polynome P)** qui retourne le degré d'un polynôme **P** (i.e. le plus grand degré des monômes le constituant).
6. Écrire une fonction réelle **valeurEn (polynome P, float X)** qui calcule la valeur d'un polynôme **P** en fonction d'une valeur réelle **x** passée en paramètre. On utilisera une fonction **puissance** définie par ailleurs.
7. Écrire une fonction **derive (polynome P)** qui calcule et retourne le polynôme dérivé de **P**.
8. Écrire une procédure **ajoutMonome (polynome & P, monome M)** qui ajoute le monôme **M** au polynôme **P**.
9. Écrire une fonction **somme (polynome P1, polynome P2)** qui calcule et retourne la somme de deux polynômes **P1** et **P2**.
10. Écrire une fonction **multMonome (polynome P, monome M)** qui calcule et retourne le produit du polynôme **P** par le monôme **M**. Considérer le cas où le coefficient de **M** est nul.
11. Écrire une fonction **produit (polynome P1, polynome P2)** qui calcule et retourne le produit de deux polynômes **P1** et **P2**.
12. Écrire une procédure **ordonne(polynome & P)** qui ordonne par degré décroissant tous les monômes composant un polynôme¹.

Exemple d'exécution permettant de tester ces sous-programmes :

```
Nombre de monomes : 4
Saisir le polynome (coeff puis degre de chaque monome) : 1 4 -4 3 -2 1 7 0
P(X) = +X4 -4X3 -2X +7
Degré : 4
Valeur de X : -1.5
P(-1) = 28.5625
P'(X) = +4X3 -12X2 -2
P(X) + P'(X) = +X4 -2X +5 -12X2
P(X) * P'(X) = +4X7 -28X6 -10X4 +48X5 +60X3 +4X -84X2 -14
Ordonné : +4X7 -28X6 +48X5 -10X4 +60X3 -84X2 +4X -14
```

¹S'inspirer de l'algorithme du tri à bulles.

2 Complexité et tris

2.1 Calcul de complexité

Écrire les deux algorithmes suivants dans le langage C++ (sous-programmes uniquement). Expliquer ce qu'ils effectuent et déterminer leur complexité.

```
Procédure P1
Donnée : T: tableau[1..N] d'entiers
Variables : Somme, Imax, Imin, i: entier
Début
Somme = T[1]
Imax = 1
Imin = 1
Pour i de 2 à N par pas +1
    Somme = Somme + T[i]
    Si T[i] > T[Imax]
        Imax = i
    Sinon Si T[i] < T[Imin]
        Imin = i
    FinSi
FinSi
Finpour
Affiche Somme, T[Imin], T[Imax]
Fin
```

```
Procédure P2
Donnée mod.: A: tableau[1..N, 1..N] d'entiers
Variables : i, j: entier
Début
Pour i de 1 à N
    A[i,i] = 0
Finpour
Pour i de 1 à N
    Pour j de (i+1) à N
        Si A[i,j] < 0
            A[i,j] = 0
        FinSi
    Finpour
Fin
```

2.2 Recherches multiples dans un tableau trié

On cherche à déterminer à partir de quel nombre k de recherches dans un tableau entier, il est préférable de passer d'une solution qui cherche dans un tableau non trié à une solution de recherche avec dichotomie dans le tableau trié.

La recherche dans un tableau non trié comportant n éléments est de complexité $O(n)$. Par contre, pour la seconde méthode de recherche, on doit d'abord trier le tableau avec une méthode de tri de complexité $O(n \times \log_2(n))$ et chaque recherche d'un élément dans le tableau utilise alors la méthode de dichotomie de complexité $O(\log_2(n))$.

Pour simplifier, on suppose les temps de calcul moyens suivants pour les différents algorithmes :

- algorithme $R_1(v, t, n)$ (recherche d'une valeur v dans un tableau t de taille n) : n unités de temps ;
- algorithme $R_2(v, t, n)$ (recherche d'une valeur v dans un tableau t de taille n) : $\log_2(n)$ unités de temps ;
- algorithme $T(t, n)$ (tri d'un tableau t de taille n) : $n \times \log_2(n)$ unités de temps ;

1. Si $n = 10^6$ et $k = 10$, indiquer quelle méthode demandera le moins de calculs.
2. En gardant $n = 10^6$, à partir de quel nombre k de recherches la seconde méthode devient plus rapide que la première ? Écrire un programme permettant de déterminer cette valeur de k .

2.3 Déplacement d'éléments particuliers dans un tableau

1. On considère un tableau de n entiers. Écrire un algorithme qui amène tous les éléments nuls en début de tableau et calcule le nombre p d'éléments nuls.
2. Calculer la complexité de l'algorithme proposé, en dénombrant les comparaisons et les affectations dans le pire des cas.
3. Envisager plusieurs méthodes, et discuter des différences en termes de résultat (le tableau modifié) et de complexité.

2.4 Tri casier

Le tri casier est un algorithme de tri pouvant s'appliquer à un tableau d'entiers dont les valeurs sont comprises dans un intervalle prédéterminé. Le tri casier consiste à compter le nombre d'occurrences de chaque valeur et est très efficace si l'intervalle des valeurs possibles est de taille raisonnable.

On considère un tableau **T** composé de **N** valeurs entières comprises entre 0 et une constante **K**.

L'algorithme procède en deux étapes :

- initialiser et remplir un tableau auxiliaire d'occurrences permettant d'enregistrer, pour chaque valeur entière **v** de $[0..K]$, le nombre d'éléments du tableau **T** qui ont pour valeur **v** ;
- utiliser ce tableau d'occurrences pour établir le tableau **T** trié.

1. Expliquer comment cette méthode permet de trier le tableau *T* suivant contenant 8 valeurs de $[0..5]$ ($N = 8$ et $k = 5$ ici):

2	5	3	0	2	3	0	3
---	---	---	---	---	---	---	---

2. Écrire cet algorithme, et calculer sa complexité.

2.5 Tris simples [TP]

Écrire les sous-programmes effectuant les traitements suivants :

1. Déterminer la position (indice) de l'élément maximum d'un tableau de n entiers ;
2. Trier un tableau de n entiers selon le principe du tri par sélection, au moyen du sous-programme précédent et d'une procédure d'échange ;
3. Insérer à la bonne position un entier x dans un tableau trié de n entiers ;
4. Trier un tableau de n entiers selon le principe du tri par insertion, au moyen du sous-programme précédent.

Tester ces deux méthodes de tri en ajoutant des sous-programmes permettant de générer aléatoirement² des tableaux d'entiers, et de vérifier s'ils sont correctement triés.

²La fonction **rand** (nécessitant la librairie **cstdlib**) retourne un nombre entier (pseudo-)aléatoire compris (au sens large) entre 0 & **RAND_MAX** (constante définie dans **cstdlib**, pouvant varier selon les compilateurs / architectures...). Initialiser ce générateur (une seule fois, avant le 1^{er} appel à **rand**) avec l'instruction **srand(time(NULL))** par exemple (librairie **ctime**).

2.6 Tri rapide, comparaison de tris [TP]

1. Implémenter et tester l'algorithme du *quicksort*.
2. Comparer ses temps d'exécution avec ceux des tris simples en $O(n^2)$ (exercice 2.5), en fonction de différentes tailles de tableau. Pour être comparés, deux algorithmes doivent être appliqués sur les mêmes données exactement.
3. Comparer ses temps d'exécutions avec ceux de l'algorithme du tri casier (exercice 2.4), pour différentes valeurs de N et de K . Expliquer les grandes variations d'efficacité de ces algorithmes selon l'ordre de grandeur de N et K .

Mesurer des temps d'exécution en C++.

Pour mesurer la durée de l'exécution d'un bloc d'instructions ou d'un sous-programme, il est possible de suivre ces étapes :

- Stocker dans une variable le moment de début. La fonction `std::chrono::system_clock::now()` renvoie une valeur de type `std::chrono::time_point` représentant le moment actuel en temps système.
L'instruction est alors : `std::chrono::time_point moment_debut = std::chrono::system_clock::now();`
ou simplement : `auto moment_debut = std::chrono::system_clock::now();`
- Effectuer l'opération à chronométrer, en l'occurrence ici exécuter le tri.
- Stocker dans une variable le moment de fin : `auto moment_fin = std::chrono::system_clock::now();`
- Calculer la différence entre le moment de fin et le moment de début pour obtenir la durée écoulée.
Par exemple, avec comme unité les millisecondes : `int duree = std::chrono::duration_cast<std::chrono::milliseconds>(moment_fin - moment_debut).count();`

Le code suivant permet donc de déterminer le temps de calcul consommé par un sous-programme `tri(T,N)` :

```
auto moment_debut = std::chrono::system_clock::now();
tri(T,n);
auto moment_fin = std::chrono::system_clock::now();
float duree_secondes = static_cast<float>(
    std::chrono::duration_cast<std::chrono::milliseconds>(moment_fin - moment_debut).count()
)/1000;
```

Ces fonctions nécessitent la bibliothèque `chrono`.