L1 Informatique

Algorithmique 2 – Exercices[1]

1 Sous-programmes

1.1 Puissance réelle

1. ⓐ Écrire une fonction réelle `puissance`, prenant en paramètres deux paramètres réels `a` et `b` (en supposant $\texttt{a} \geq 0$), basée sur la définition suivante[2] :

$$\begin{cases} 0^b = 0 & ,\forall b \in \mathbb{R} \\ a^b = e^{b.\ln(a)} & ,\forall (a,b) \in \mathbb{R}_+ \times \mathbb{R} \end{cases}$$

2. ⓐ Écrire un programme principal utilisant cette fonction. Attention à la validité des arguments.

3. ⋆ Écrire deux fonctions permettant de calculer une valeur approchée des fonctions exponentielle et logarithme népérien, à partir des formules suivantes :

   - $e^x = \sum_{k=0}^{+\infty} \frac{x^k}{k!}$
   - $\ln(x) = 2 \cdot \sum_{k=0}^{+\infty} \frac{1}{2k+1} (\frac{x-1}{x+1})^{2k+1}$

   Définir pour cela une constante de précision qui déterminera le nombre de termes de la somme infinie à calculer.

4. ⓢ [TP] Comparer les résultats de puissances entières calculées en utilisant ou non les fonctions de la bibliothèque `cmath`. Ajuster si besoin la constante de précision.

1.2 Triangles [TP]

Cet exercice consiste à écrire un programme permettant de vérifier les propriétés des triangles en fonction de leurs mesures.

1. Écrire une procédure qui permet de saisir la longueur des trois côtés d'un triangle.

2. Écrire une fonction qui détermine la validité d'un triangle en fonction de ses trois côtés.

3. Écrire une fonction qui calcule le périmètre d'un triangle.

4. Écrire une fonction qui calcule l'aire d'un triangle à partir des longueurs a, b, c de ses côtés, en utilisant la formule de Kahan : $\text{Aire} = \sqrt{s \cdot (s-a) \cdot (s-b) \cdot (s-c)}$, avec $s = (a+b+c)/2$.

5. Écrire quatre fonctions qui déterminent si un triangle est respectivement équilatéral, isocèle, rectangle, plat.

6. En utilisant les fonctions précédentes, écrire une fonction qui retourne la chaine de caractères correspondant à la nature d'un triangle (par exemple : `triangle équilatéral`, `triangle isocèle plat`, `triangle quelconque`, *etc.*).

[1] Légende : ⓐ Application directe du cours. [TP] Exercice de TP à tester sur machine. ⓢ Exercice supplémentaire (travail personnel). ⋆ Exercice ou question plus difficile. Dans les exemples d'exécution, les valeurs en gras indiquent des lectures au clavier.

[2] Les fonctions exponentielle et logarithme népérien sont utilisables via la bibliothèque **`cmath`** (fonctions **`exp`** et **`log`**).

7. Compléter le programme de manière à utiliser les fonctionnalités précédentes comme dans l'exemple ci-dessous :

```
Longueur des côtés : 2.2 8 3
Triangle invalide, recommencez.
Longueur des côtés : 5 3.5 5
Il s'agit d'un triangle isocèle de périmètre 13.5 et d'aire 8.2
```

2 Sous-programmes et tableaux

2.1 Liste de nombres

On définit le type liste de la manière suivante :

```
const int taillemax = 1000 ;
using liste = std::array <float, taillemax>;
```

1. Écrire une procédure permettant de saisir une liste de `n` nombres réels.
2. Écrire une procédure permettant d'afficher une liste de `n` nombres réels, séparés par des points-virgules.
3. Écrire une fonction booléenne qui détermine si une liste de `n` nombres réels est triée par ordre croissant.
4. Écrire une fonction réelle calculant le plus grand écart séparant deux valeurs consécutives d'une liste de nombres réels.
5. En utilisant les sous-programmes précédents, écrire un programme principal qui demande à l'utilisateur une suite de nombres, l'affiche, et indique si elle est triée par ordre croissant ou non. Si la liste est correctement triée, le programme affichera le plus grand écart séparant deux valeurs consécutives.

2.2 Statistiques

On veut calculer la moyenne et la variance de l'âge des individus d'un groupe. Pour cela, on utilisera un tableau qui contiendra, pour un âge donné (en indice), le nombre d'individus correspondant.

1. Définir un type TabAges.
2. Écrire une procédure `init` qui initialise un tableau d'âges (aucun individu).
3. Écrire une procédure `saisie` qui permet à un utilisateur de saisir une série d'âges.
4. Écrire une procédure `regroupe` qui additionne deux tableaux d'âges.
5. Écrire une fonction `moyenne` qui calcule et renvoie l'âge moyen d'un tableau d'âges.
6. De la même manière, écrire les fonctions `variance` et `ecarttype`.
7. Écrire une fonction `median` qui renvoie l'âge médian d'un tableau d'âges.

2.3 Précipitations [TP]

On s'intéresse à l'étude des précipitations pour chaque mois de l'année pour les villes d'Angers, de Nantes et de Rennes. Les différentes valeurs (entières) seront conservées dans un tableau pour chaque ville.

1. Définir le type `TabPrecipitations`.
2. Écrire une procédure de saisie et une procédure d'affichage d'un tableau de précipitations.

3. Écrire les fonctions `minimale` et `maximale` qui calculent respectivement la plus petite et la plus grande valeur d'un tableau de précipitations.

4. Compléter le programme de manière à déterminer la plus grande et la plus petite valeur de précipitations sur l'ensemble des villes. Des fonctions annexes pourront être définies.

2.4 Séparation et fusion de tableaux

1. Écrire une procédure de saisie et une procédure d'affichage d'un tableau `T` de `n` entiers.

2. Écrire une procédure qui, à partir d'un tableau `T` de `n` entiers non nécessairement ordonnés, et d'une valeur entière `x`, construit deux tableaux `U` et `V` tels que `U` soit composé de toutes les valeurs de `T` strictement supérieures à `x`, et `V` des valeurs de `T` inférieures ou égales à `x`.

3. Écrire une procédure qui, à partir de deux tableaux triés, crée un troisième tableau trié contenant l'ensemble des éléments des deux premiers tableaux.

2.5 Représentation d'un ensemble par un tableau

On utilise un tableau pour stocker un ensemble de nombres. Un tableau code un ensemble si et seulement si le même élément n'appartient pas plusieurs fois au tableau.

1. Écrire une fonction `occurrence` qui fournit le nombre d'éléments d'une valeur donnée appartenant à un tableau `T` composé de `n` valeurs entières.

2. Écrire une fonction `appartient` qui détermine si une valeur donnée appartient à un tableau.

3. Écrire une fonction `ensemble qui` permet de savoir si un tableau `T` code bien un ensemble.

4. Écrire une procédure `devient_ensemble` qui, à partir d'un tableau `T1` de `n1` entiers, construit un tableau `T2` de `n2` entiers contenant une et une seule fois chaque valeur présente dans `T1`.

5. Écrire deux procédures `inter_ens` et **`union_ens`** qui calculent respectivement l'intersection et l'union de deux ensembles.

2.6 Saute-moutons [TP]

Le jeu du saute-moutons se joue avec `M` moutons noirs et `M` moutons blancs positionnés en ligne. On les représente dans un tableau de caractères de taille `2M+1` (pour les tests, prendre `M=3`). Le caractère `B` représente un mouton blanc, `N` un mouton noir, et `X` un emplacement libre.

On initialise le jeu ainsi :

B	B	B	X	N	N	N
0	1	2	3	4	5	6

Le but du jeu est d'inverser les positions des moutons blancs et noirs, sachant que :

- les moutons blancs ne peuvent avancer que vers la droite ;
- les moutons noirs ne peuvent avancer que vers la gauche ;
- un mouton peut avancer si la case suivante est libre (exemple 2 vers 3, ou 4 vers 3) ;
- un mouton peut sauter si la case suivante est un autre mouton et celle d'après est libre (exemple 1 vers 3 ou 5 vers 3) ;
- si plus aucun mouton ne peut bouger, la partie est perdue ;
- si les moutons ont inversé leur position, la partie est gagnée.

Partie gagnée :

N	N	N	X	B	B	B
0	1	2	3	4	5	6

Exemple de partie bloquée (perdue) :

B	B	N	B	N	N	X
0	1	2	3	4	5	6

1. Définir le type **jeu** (tableau de caractères).
2. Écrire une procédure **init** qui initialise le plateau de jeu.
3. Écrire une procédure **afficher** qui affiche le plateau de jeu. À l'affichage, l'emplacement libre sera matérialisé par un espace.
4. Écrire une fonction **gagne** qui renvoie *vrai* si la partie est gagnée, et *faux* sinon.
5. Écrire une procédure **joue** qui demande à l'utilisateur le mouton à déplacer, et le déplace si cela est possible. On pourra décomposer le problème en écrivant une fonction **caseVide** qui retourne l'emplacement de la case vide, et une fonction **estJouable** qui retourne *vrai* si l'on peut déplacer le mouton d'une case **c** vers la case vide).
6. Finaliser le programme de manière à ce que l'utilisateur joue jusqu'à ce que la partie soit gagnée. Pour améliorer le jeu, on veut terminer la partie si aucun coup n'est jouable. Écrire une fonction **perdu** qui détermine si aucun coup n'est possible. Modifier le programme principal en conséquence.

3 Sous-programmes et chaines de caractères

3.1 Conversion d'une chaine de caractères en nombre

1. Écrire les fonctions **caractereEntier** et **entierCaractere** permettant d'attribuer à tout caractère représentant un chiffre, une valeur numérique, et réciproquement.
   → Par exemple, `caractereEntier('6')` vaut 6, et `entierCaractere(0)` vaut '0'.
2. Écrire deux fonctions **chaineEntier** et **entierChaine** permettant la conversions d'une chaine de caractères en un nombre entier, et vice versa.

3.2 Autres fonctions sur les chaines [TP]

Écrire les sous-programmes suivants permettant de manipuler les chaines de caractères :

1. **copie(s,i,j)** : fonction qui retourne de la chaine de caractère **s**, un nombre **j** de caractères à partir de la position **i** (dans le sens de la lecture).
2. **supprime(s,i,j)** : procédure qui supprime dans la chaine nommée **s**, un nombre **j** de caractères à partir de la position **i**.
3. **insere(s1,s2,i)** : procédure qui insère la chaine **s1** dans la chaine **s2** à la position **i**.
4. **position(s1,s2)** : fonction qui renvoie la position de la chaine **s1** dans la chaine **s2** (si **s1** n'est pas incluse dans **s2**, alors cette fonction renvoie la valeur -1).

3.3 ⋆ Algorithme de l'addition Ⓢ

Écrire un programme permettant d'additionner deux nombres entiers enregistrés sous forme de chaines de caractères. Il n'est pas permis d'utiliser de variables numériques (`int`), donc pas de conversion ou d'addition, excepté pour le parcours des chaines de caractères (variable d'indice et incrémentation de cette variable).

4 Sous-programmes et tableaux à deux dimensions

4.1 Matrice creuse Ⓢ

Une matrice creuse est une matrice composée d'un grand nombre de zéros. Le pourcentage de valeurs nulles dans une matrice permet de déterminer à quel point une matrice est creuse ou au contraire dense. On considère une matrice non nécessairement carrée à valeurs entières, représentée par un tableau à deux dimensions.

1. Écrire des procédures de saisie et d'affichage d'une matrice dont le nombre de lignes et le nombre de colonnes sont donnés en paramètres.
2. Écrire une fonction calculant la proportion de zéros dans une matrice passée en paramètre.
3. Écrire une procédure permettant de générer aléatoirement une matrice binaire (composée de 0 et 1) dont la proportion de 0 est donnée en paramètre.

4.2 Carré magique Ⓢ

Écrire un programme permettant à un utilisateur de saisir une grille carré de $n \times n$ valeurs, et de vérifier s'il s'agit d'un carré magique d'ordre n. Un carré magique comporte tous les nombres de l'intervalle $[1..n^2]$, et la somme des valeurs de toute ligne, colonne et diagonale doit toujours être égale à la même valeur $n(n^2+1)/2$.

4.3 Jeu du morpion [TP] Ⓢ

On souhaite écrire un programme permettant à deux personnes de jouer au morpion. Pour commencer, on ne considère que les alignements sur les lignes et les colonnes, et pas sur les diagonales. Le pion d'un joueur est représenté par un caractère : `'X'` ou `'O'`. La grille est représentée par un tableau de caractères (espace pour les cases libres, `'X'` ou `'O'` pour les cases occupées par l'un ou l'autre joueur). Les joueurs sélectionnent à tour de rôle une case de la grille. Le jeu s'arrête lorsque l'un des joueurs a réalisé un alignement de 4 pions ou bien lorsque la grille est remplie sans que personne n'ait gagné.

L'algorithme à implanter est le suivant :

```
Initialisations (taille de la grille, grille vide, premier joueur, etc.)
Répéter
    Changer de joueur
    Faire jouer le joueur courant
    Tester s'il a gagné
    Afficher la grille
Jusqu'à Fin du jeu
Afficher le résultat
```

Écrire les sous-programmes qui :

1. Change le joueur courant.
2. Lit le coup du joueur courant (numéros de ligne et de colonne), vérifie sa validité, et met à jour la grille. On écrira une fonction qui teste la validité d'un coup (la case jouée appartient à la grille et n'est pas déjà occupée).
3. Teste si une ligne (resp. une colonne) contient un alignement de 4 pions du joueur courant.
4. Teste si le jeu est fini.

→ ⋆ Compléter le programme afin qu'il puisse également tester les diagonales.

5 Les fichiers

5.1 Comparaison de fichiers "texte" [TP]

1. Créer manuellement 3 fichiers `fic1.txt`, `fic2.txt` et `fic3.txt` tels que `fic2` soit une simple copie de `fic1` et `fic3` soit différent de `fic1`.
2. Écrire un sous-programme à 2 arguments (2 noms de fichier) qui teste si les 2 fichiers sont identiques en utilisant la fonction `getline`. Le sous-programme renverra "Oui" en cas de correspondance (e.g. avec `fic1.txt` et `fic2.txt`), "Non" en cas de différence (`fic1.txt` et `fic3.txt`) et "Non applicable" en cas de problème d'accès aux fichiers (e.g. `fic1.txt` et un autre fichier inexistant).
3. Même question que la précédente MAIS en utilisant `get`.
4. Est-il possible de faire de même en autorisant la lecture **exclusivement** via l'opérateur >> ? Justifier.
5. Écrire un (sous-)programme "principal" permettant de tester les 2 (ou 3 si réponse affirmative à la question précédente) sous-programmes.

5.2 Notes Algo1-P2

On suppose l'existence d'un fichier texte "notes.txt" contenant sur chaque ligne le nom d'un étudiant de L1 suivi de sa note en Algo1–P2, les 2 informations étant séparées par (au moins) un espace.

1. Écrire un programme permettant d'afficher la moyenne des notes stockées dans "notes.txt" et d'identifier le major de promotion.
2. Compléter le programme afin de créer un fichier "notes_apres_jury.txt", similaire à "notes.txt" **mais** où la note de chaque étudiant a été augmentée de 0.5.

5.3 Une ébauche de la commande UNIX `wc` [TP]

Écrire un programme qui affiche les nombres de lignes, caractères (sans les espaces, tabulations ...[3]) et mots contenus dans un fichier texte en ne faisant qu'une seule lecture du fichier.

[3] `std::isspace(c)` renvoie un entier strictement positif si le caractère `c` est un espace, une tabulation ou un retour à la ligne, et `0` sinon