R1.01 – TD6

Exercice 1 : écriture de prédicats

La fonction `trouveCarInStr ()` recherche la présence (ou l'absence) d'un caractère dans un ensemble. Or si l'ensemble est continu (il ne contient pas de trou) ou s'il est très petit (3,4 éléments aux maximum), nous n'avons pas besoin de parcourir tout cet ensemble, seules quelques comparaison suffisent.

Ecrire les prédicats suivants :

```
fonction isUpper (car : in caractere) renvoie booleen;
fonction isLower (car : in caractere) renvoie booleen;
fonction isDigit (car : in caractere) renvoie booleen;
```

Ces prédicats renvoient `vrai` si le caractère qui leur est passé en paramètre est respectivement une majuscule, une minuscule ou un chiffre décimal (de `'0'` à `'9'`).

Ecrire le prédicat `isSpace ()`, qui renvoie vrai si le caractère passé en paramètre est un caractère d'espacement (espace, tabulation ou retour à la ligne).

Exercice 2 : comptage de doublons dans une chaîne de caractères - V1

Une catégorie d'algorithmes, plus complexe, consiste à appliquer à chaque élément un traitement **qui dépend du contexte** de cet élément dans le tableau, par exemple de la valeur de l'élément qui le précède ou qui le suit, etc. C'est à ce type d'algorithme que correspond l'exercice proposé ici.

L'algorithme demandé doit saisir (au clavier bien sûr !) une suite de lignes (des `string`s), jusqu'à ce que l'utilisateur tape une ligne vide (appuie sur la touche Entrée sans avoir saisi aucun caractère).

Un traitement (précisé plus loin) doit être effectué sur les **doublons** de chaque ligne saisie (un doublon = deux caractères "consécutifs" identiques, définition précisée plus loin). Ce peut être par exemple un comptage.

L'algorithme de haut niveau est évident :

```
algorithme compterDoublons
debut

    boucle
        declarerEtSaisirUneLigne;
        si (ligneLueEstVide) sortie;

        // La ligne traitée est non vide
        effectuerLeTraitement; // comptage, en balayant la ligne
        afficherLeResultat;
    fboucle

fin
```

La phase suivante du développement consiste :

- soit à remplacer directement les trois actions `declarerEtSaisirUneLigne`, `effectuerLeTraitement` et `afficherLeResultat` par les instructions qui les composent,
- soit à développer les trois actions sous forme de sous-programmes, l'algorithme de haut niveau se contentant de les appeler,
- soit de réaliser un compromis entre les deux solutions précédentes.

C'est cette dernière solution que nous proposons ici :

- `declarerEtSaisirUneLigne` et `afficherLeResultat` ne présentent pas de difficulté, et sont suffisamment simples pour être directement intégrées à l'algorithme de haut niveau,
- en revanche, `effectuerLeTraitement` est plus complexe, et mérite donc d'être isolé sous forme d'un sous-programme, plus précisément une fonction qui renvoie le résultat.

Travail demandé

Ecrire la fonction `compterDoublons()` présentée ci-dessous. Les "règles du jeu" (les spécifications) sont les suivantes :

1. un doublon est une suite de deux caractères consécutifs (deux éléments dont les rangs diffèrent d'une unité) identiques.
2. **trois** caractères consécutifs identiques constituent **deux** doublons.

Les doublons de caractères d'espacement **ne sont pas comptabilisés**.

Lorsque le caractère courant est de rang `i`, il y a deux façons de tester s'il appartient à un doublon :

1. soit en testant le caractère de rang précédent (de rang `i - 1`), dans ce cas, le premier élément ne peut être comparé avec son prédécesseur. Il faut donc commencer les comparaisons à l'élément de rang `1`.
2. soit en testant le caractère de rang suivant (de rang `i + 1`), dans ce cas, le dernier élément ne peut être comparé avec son successeur. Il faut donc terminer les comparaisons par l'avant-dernier élément.

Ecrire une version de la fonction `compterDoublons()` en utilisant une boucle `pour`, puis une autre en utilisant une boucle `tant_que`.

Variantes

Quelques modifications, qui peuvent paraître mineures, dans les spécifications, peuvent nécessiter des modifications profondes de l'algorithme, comme par exemple l'obligation de changer de schéma itératif.

Ecrire les modifications nécessitées par les spécifications suivantes :

Variante 1

1. un doublon est une suite de deux caractères consécutifs identiques.
2. un même caractère ne peut appartenir à deux doublons différents. En conséquence, il faut **quatre** caractères consécutifs identiques pour constituer **deux** doublons.
3. les doublons de caractères d'espacement **ne sont pas comptabilisés**

Variante 2

1. un doublon est une suite de deux caractères consécutifs identiques.
2. **trois** caractères consécutifs identiques constituent **deux** doublons.
3. les doublons de caractères d'espacement **sont comptabilisés** : par exemple un espace suivi d'une tabulation (ou l'inverse) est un doublon.

Variante 3

1. un doublon est une suite de deux caractères consécutifs identiques.
2. **trois** caractères consécutifs identiques constituent **deux** doublons.
3. les caractères d'espacement doivent être ignorés :
    - il ne faut pas compter les doublons de caractères d'espacement,
    - deux caractères identiques séparés par un nombre quelconque de caractères d'espacement, forment un doublon.

Exercice 3 : comptage de doublons dans une chaîne de caractères - V2

L'approche envisagée ici en est totalement différente : l'algorithme de haut niveau proposé consiste à se positionner successivement sur les doublons successifs, et à les comptabiliser au fur et à mesure. L'algorithme se termine lorsque plus aucun doublon n'est trouvé.

L'analyse est rigoureusement analogue à celle qui a été étudiée dans "Fonction de comptage d'un caractère dans une chaîne de caractères". Elle passe par l'écriture de la fonction `findFirstDoublon()`, de profil :

```
fonction findFirstDoublon (chaine : in string,
                           debut  : in entier_naturel)
    renvoie entier_naturel;
```

qui renvoie la position du **premier** caractère du **premier** doublon trouvé dans `Chaine` **à partir** de la position initiale `debut`. Elle renvoie `taille (chaine)` si aucun doublon n'est trouvé.

Travail demandé

Ecrire le corps de la fonction `findFirstDoublon()`, puis écrire l'algorithme qui compte le

nombre de doublons dans une chaîne saisie au clavier.

Remarques

1. prendre n'importe laquelle des définitions des doublons de l'exercice précédent;
2. la ligne peut être vide.

Exercice 4 : recherche d'une sous-chaîne dans une chaîne de caractères

Comme on l'a vu dans l'exercice "Comptage de doublons dans un tableau - V2", le comptage d'un doublon (couple de lettres consécutives identiques) peut être considéré comme la répétition de recherches successives de ce doublon à partir de la position du doublon précédemment trouvé.

Un doublon peut être considéré comme un **motif** (***pattern***) connu par sa **propriété**.

On peut envisager des motifs définis par leur **valeur**, exemple "Comptage des suites le, les, lle".

La forme générale de l'algorithme est totalement identique, seule change la reconnaissance du **motif**.

Lorsque le motif dépasse deux caractères, il devient lourd et maladroit de mémoriser individuellement les caractères qui composent le motif. Il est préférable de considérer que l'on cherche une sous-chaine (de 2, 3 lettres ou plus) dans une chaîne (ou un sous-tableau dans un tableau). C'est ce qui vous est proposé dans l'exercice ci-dessous.

Ecrire la fonction `findSubstrInStr()` qui renvoie le rang de la première apparition d'une sous-chaîne dans une chaîne de caractères, à partir d'un rang de début de recherche, tous trois passés en paramètres. Plus précisément, elle renvoie le rang du premier caractère de la sous-chaîne dans la chaîne.

La valeur de retour est obligatoirement dans l'intervalle `[0, taille (chaine) - 1]` si la sous-chaîne est présente. On choisira donc de renvoyer la valeur `taille (chaine)` si la sous-chaîne n'a pas été trouvée.

Ecrire l'algorithme qui saisit au clavier une ligne, puis dans une boucle :

- saisit une sous-chaîne jusqu'à une sous-chaîne vide,
- l'affiche **entre guillemets**, suivie de son rang dans la chaîne si elle est présente, ou d'un message indiquant qu'elle n'a pas été trouvée. Utiliser évidemment la fonction `findSubstrInStr()` !

La solution qui est demandée ici est la plus simple : la sous-chaîne est comparée à la chaîne à partir de la première position de la chaîne. Si la coïncidence n'est pas totale, elle est de nouveau comparée à la chaîne à partir de la deuxième position, etc.