Séance 4: Tableaux d'entiers et de caractères

Université Paris Cité

Objectifs:

- Manipuler des tableaux unidimensionnels d'entiers et de caractères.
- Trier un tableau d'entiers par comptage.

Exercice 1 (Des ensembles,  , ⋆)

1. *Écrire une fonction* `search` *qui prend en argument un tableau d'entiers* `tab` *et un entier* `x` *et qui renvoie* `true` *si* `tab` *contient la valeur* `x`*, et* `false` *sinon.*

   Contrat:

   ```
   int[] tab={6,20,12,1000,8}
      System.out.print(search(lis, 12)) affiche true
      System.out.print(search(lis, 50)) affiche false
   ```

2. *Écrire une fonction* `union` *qui prend en argument deux tableaux d'entiers* `tab1`*, et* `tab2` *(considérés sans doublons) et qui renvoie l'union de* `tab1` *et* `tab2`*.*

   Contrat:

   ```
   tab1={6,20,12,1000,8}, tab2={2,8,6,7,12}
   union(tab1, tab2) renvoie {6,20,12,1000,8,2,7}
   ```

□

Exercice 2 (Décalage, ⋆⋆)

Écrire une fonction `shift` *qui prend en argument un tableau d'entiers* `tab` *et qui renvoie une copie de ce tableau où toutes les valeurs sont décalées d'une case vers la droite (et la dernière valeur se retrouve en position 0).*

Contrat:

```
int t[] = {1000,1,2,3}
shift(t) doit renvoyer le tableau {3,1000,1,2}.
```

Même chose mais avec un décalage de `n` *cases (où* `n` *est un nouveau paramètre de la fonction). Que faire quand* `n` *est négatif ?* □

Exercice 3 (Fibonacci (bis), ⋆⋆)

La suite de Fibonacci $(F_n)_{n\geq 1}$ *est definie par* $F_0 = 0$, $F_1 = 1$ *et* $F_{n+2} = F_{n+1} + F_n$. *Écrire une fonction* `fibonacci` *qui renvoie un tableau de taille* n *contenant les premiers termes de la suite.*

Contrat:

```
fibonacci(5) doit renvoyer {0,1,1,2,3}.
```

□

Exercice 4 (Tableaux de caractères, ⋆)

1. *Écrire une procédure* `letters2word` *qui prend en argument un tableau de caractères* `tab` *et qui affiche le mot obtenu en concatenant ces lettres.*

   Contrat:
   `char[] tab={'p','l','a','c','a','r','d'}`
   `letters2word(tab)` *doit afficher "placard"*

2. *Écrire une procédure* `stutterword` *qui prend en argument un tableau de caractères* `tab1` *et un tableau d'entiers* `tab2` *et qui affiche le mot obtenu en concatenant les lettres du tableau* `tab1`*, comme suit : la lettre sur la position* `i` *dans* `tab1` *est répétée autant de fois que l'indique le numéro sur la position* `i` *dans* `tab2`*. La procédure doit afficher "Erreur" si les deux tableaux n'ont pas la meme longueur.*

   Contrat:
   `char[] tab1={'a','b','c','d'}`
   `int[] tab2={2,2,3,4}`

   `stutterword(tab1,tab2)` *affichera "aabbcccdddd"*

3. *Écrire une fonction* `word2letters` *qui prend en argument un mot et renvoie le tableau de ses lettres.*

   Contrat:
   `word2letters`*("placard") renvoie* `{'p','l','a','c','a','r','d'}`

4. *Écrire une fonction* `letters` *qui prend en argument un mot (chaine de caractères)* `word` *et qui renvoie le tableau de ses lettres, cette fois ci sans doublons.*

   Contrat:
   `letters`*("electroacoustique") renvoie* `{'e','l','c','t','r','o','a','u','s','i','q'}`

□

Exercice 5 (Tableaux d'entiers aléatoires,  , ⋆)

Le but de cet exercice est de mettre en œuvre des opérations simples sur des tableaux d'entiers, aboutissant à une méthode de tri particulièrement simple et efficace pour des tableaux contenant des entiers "pas trop grands". Vous allez compléter le fichier `RandomArray.java`*. Les tests seront à placer dans la fonction* `main` *de ce fichier. Vous disposez de la fonction* `boolean IntArrayEquals(int[] a, int[] b)` *qui vérifie si ses deux arguments sont égaux (même longueur, mêmes éléments rang par rang), et de la procédure* `void printIntArray (int[] a)` *qui affiche le contenu du tableau* `a`*.*

1. *Écrire une fonction* `int[] createRandomArray(int n)` *qui renvoie un tableau d'entiers de taille* `n`*, dont les cases sont remplies par des entiers aléatoires compris entre* `0` *et* `(n-1)`*. L'appel de fonction* `rand.nextInt(n)` *renvoie de tels entiers. Pour tester la fonction, supprimez les commentaires des quatre premières lignes du* `main`*.*

2. *Écrire une fonction* `int[] minMaxAverage(int[] a)` *qui renvoie un tableau de taille 3 contenant dans sa première case le minimum, dans sa deuxième le maximum et dans sa troisième la partie entière de la moyenne des éléments de* `a`*.*

   Contrat:
   L'appel
   `printIntArray(minMaxAverage(createRandomArray(100)))`
   peut afficher, par exemple,
   `0 96 46`
   Les valeurs exactes sont imprévisibles pour un tel appel, à cause du remplissage aléatoire.

3. *Écrire une fonction* `int[] occurrences(int[] a)` *qui renvoie un tableau contenant, à la case d'indice* `i`*, le nombre d'occurrences de* `i` *dans* `a`*. On supposera que les entiers contenus dans* `a` *sont positifs ou nuls. La taille du tableau* `occurrences(a)` *est* `1+(minMaxAverage(a))[1]`.

   Contrat:
   Si `a` *est le tableau* `{1,3,0,0,0,1}`, `occurrences(a)` *renvoie* `{3,2,0,1}`,

4. *Une fois obtenu le tableau* `occurrences(a)`*, il est possible de trier*[1] `a` *très simplement : en commençant à l'indice* `0`*, on insère autant de* `0` *qu'indiqué dans la première case du tableau des occurrences, puis autant de* `1` *qu'indiqué dans sa deuxième case et ainsi de suite. Appliquée à l'exemple précédente, cette méthode produit le tableau* `{0,0,0,1,1,3}` *(d'abord 3* `0`*, puis 2* `1`*, puis 0* `2`*, puis 1* `3`*), qui est bien ce qu'on voulait.*

   (a) *Écrire une fonction* `int[] countingSort(int[] a)`*, qui utilise* `occurrences`*, et renvoie un tableau trié contenant les mêmes éléments que* `a` *en utilisant l'algorithme décrit ci-dessus*[2].

   (b) *Écrire une procedure* `void countingSort2(int[] a)`*, qui utilse le même procédé que* `countingSort` *pour trier* `a` sur place *: le contenu de* `a` *change suite à l'appel* `countingSort2(int[] a)`.

   Contrat:

   L'exécution de :
   ```
   int[] a = createRandomArray(100);
   int[] b = countingSort(a);
   countingSort2(a);
   System.out.println(intArrayEquals(a,b));
   ```
   affiche `true`.

□

Exercice 6 (Tri, 🔍, ⋆⋆- ⋆⋆⋆)

1. *Écrire une fonction* `position` *qui prend en argument un tableau d'entiers* `tab`*, qu'on considére déjà trié, et un entier* `x` *et qui renvoie la position dans* `tab` *dans laquelle on devrait insérer* `x`*, pour que le tableau obtenu reste trié.*

   Contrat:
   `int[] tab={0,2,4,6,7,8}`
   `position(tab,1)` *renvoie 1*
   `position(tab,-5)` *renvoie 0*
   `position(tab, 10)` *renvoie 6*

2. *Écrire une fonction* `insert` *qui prend en argument un tableau d'entiers* `tab`*, et deux entiers,* `pos` *et* `x`*, et qui renvoie le tableau obtenu en insérant l'element* `x` *dans* `tab` *sur la position* `pos`*. Si* `pos` *est plus grand que la taille de* `tab` *, la fonction renvoie le tableau sans modification.*

   Contrat:
   `int[] tab={2,5,4,3}`
   `insert(tab, 0, 1)` *renvoie* `{1,2,5,4,3}`
   `insert(tab, 2, 100)` *renvoie* `{2,5,100,4,3}`

3. *Écrire une fonction* `sort` *qui prend en argument un tableau* `tab` *et qui renvoie le tableau trié. (Pour cela, pensez a utiliser les fonctions* `position` *et* `insert`*)*

   Contrat:
   `int[] tab = {40,1,20,3,8,6}`
   `sort(tab)` *renvoie* `{1,3,6,8,20,40}`

□

1. Trier un tableau d'entiers `a` consiste en la construction d'un tableau qui contient les mêmes éléments que `a`, disposés en ordre croissant : l'entier contenu dans la case d'indice `i` n'est pas plus grand que l'entier contenu dans la case d'indice `(i+1)`, pour tout indice.

2. Pour les tableaux engendrés par `createRandomArray(n)` ce procédé est efficace car ils ne contiennent que des entiers allant de `0` à `(n-1)`. Pour des tableaux quelconques, la taille potentiellement très grande du tableaux des occurrences peut rendre cette méthode très inefficace.