université PARIS DIDEROT PARIS 7

INTRODUCTION À L'INFORMATIQUE ET À LA PROGRAMMATION (IF1)

TP7 : TABLEAUX

1 Préambule

► **Exercice 1** : Écrire une fonction `chaine` qui prend en argument un tableau d'entiers et renvoie une chaîne de caractères représentant son contenu.

► **Exercice 2** : Écrire une fonction `saisie` qui lit un entier `n`, puis `n` entiers (les lectures correspondent à des saisies au clavier) et renvoie un tableau contenant ces `n` entiers.

► **Exercice 3** : Écrire une fonction `tableauRenverse` qui prend en argument un tableau d'entiers et renvoie un nouveau tableau de même contenu mais dans un ordre renversé.

► **Exercice 4** : Écrire une fonction `concatenation` qui prend en arguments deux tableaux d'entiers et renvoie un troisième tableau dont le contenu est celui du premier suivi de celui du deuxième.

2 Recherche

► **Exercice 5** : Écrire une fonction `indiceLin` qui prend en arguments un tableau d'entiers et une valeur entière et renvoie le plus petit indice d'occurrence de cette valeur dans le tableau, ou `-1` si un tel indice n'existe pas.

► **Exercice 6** : Écrire une fonction `estTrie` qui prend en argument un tableau d'entiers et teste si ce tableau est trié en ordre croissant.

► **Exercice 7** : Lorsque le tableau dans lequel on recherche une valeur est trié, il est plus efficace d'envisager un algorithme de *recherche par dichotomie* plutôt que la *recherche linéaire* utilisée dans l'exercice **5**.
La recherche par dichotomie manipule trois entiers $\texttt{p} \leq \texttt{m} \leq \texttt{g}$ (pour « petit », « moyen » et « grand »).
À tout moment, on a $\texttt{a[p]} \leq \texttt{v} \leq \texttt{a[g]}$ pour un tableau `a` contenant la valeur `v` à trouver.
Initialement, `p` et `g` correspondent aux indices de la première et la dernière case du tableau `a`.
À chaque étape du calcul, `m` prend la valeur de la moyenne de `p` et `g`. Si `a[m]` vaut `v`, alors l'algorithme termine. Dans le cas contraire, pour $\texttt{a[m]} < \texttt{v}$, alors `p` prend la valeur de `m+1`, sinon `g` prend la valeur de `m-1`.
L'algorithme termine pour $\texttt{p} \geq \texttt{g}$.
Voici un exemple où on recherche la valeur `42` :

1	12	18	35	42	**97**
↑p		↑m			↑g
1	12	18	**35**	42	**97**
			↑p	↑m	↑g
1	12	18	35	**42**	97

Écrire une fonction `indiceDich` qui prend en arguments un tableau d'entiers supposé trié (il n'est pas demandé de le vérifier) et une valeur entière, et qui utilise une recherche par dichotomie pour trouver un indice d'occurrence de la valeur dans le tableau (`-1` si un tel indice n'existe pas).

► **Exercice 8** : Déduire des fonctions définies aux exercices **5**, **6** et **7**, une fonction `indice` qui prend en arguments un tableau d'entiers et une valeur entière et renvoie un indice d'occurrence de la valeur dans le tableau (`-1` si un tel indice n'existe pas) via une recherche linéaire ou dichotomique selon le cas.

3 Sur place

► **Exercice 9** (Tri par insertion) : Le *tri par insertion* est un algorithme qui permet de trier un tableau *en place*, c'est-à-dire sans se servir d'un tableau additionnel. Le tri par insertion d'un tableau `a` de taille `n` utilise une variable `m` qui scinde le tableau en deux parties : les éléments d'indice `0` à `m-1` forment la partie déjà triée (fond gris sur l'exemple ci-dessous) et ceux d'indice `m` à `n-1` forment la partie restant à trier (fond blanc).
Initialement, `m` vaut `1`.
À chaque étape, on compare l'élément `a[m]` à chacun des éléments `a[0]`, ..., `a[m-1]` successivement. Lorsqu'on trouve le plus petit indice `k` avec $0 \leq k \leq m-1$ tel que `a[k]` est supérieur à `a[m]`, on insère `a[m]` « avant » `a[k]` en effectuant une permutation circulaire des éléments `a[k]`, ..., `a[m-1]`, `a[m]` pour obtenir `a[m]`, `a[k]`, ..., `a[m-1]`. Si aucun indice `k` ne convient, on laisse `a[m]` à sa place. On incrémente alors `m`.
L'algorithme termine lorsque `m` vaut `n`.

5	3	1	6	4
3	5	1	6	4
1	3	5	6	4
1	3	5	6	4
1	3	4	5	6

Exemple de tri par insertion. Dans la troisième ligne, par exemple, `m` vaut `3`, et les trois premiers éléments du tableau sont triés. Comme `6` > `5`, aucun `k` ne convient, et `6` reste en place, puis on continue avec `m` valant `4`.

Écrire une fonction `trier` qui prend un tableau d'entiers en argument et le trie en place. Vérifier qu'elle fonctionne correctement lorsque la même valeur apparaît plusieurs fois dans le tableau donné en entrée.

► **Exercice 10** : On s'intéresse à une des nombreuses variantes du jeu de l'Awélé (ou Awalé), dont le plateau est composé d'une suite de cases qui contiennent des graines. Ce dernier sera codé par un tableau dont chaque élément indiquera le nombre de graines contenues dans la case correspondante.

1. Écrire une fonction `alea` qui prend en arguments deux entiers positifs `n` et `m` et renvoie un tableau de `n` entiers positifs pseudo-aléatoires dont la somme est `2*m`.
2. Écrire une fonction `repartirFin` qui prend en arguments un tableau d'entiers et un indice `k` (supposé valable), enlève les graines de la case d'indice `k` et les répartit dans les cases suivantes à raison d'une graine par case. Si on arrive à la fin du tableau et qu'il reste des graines à répartir, on les jette.
3. Écrire une fonction `repartirTour` identique à la précédente, sauf que si on arrive à la fin du tableau et qu'il reste des graines à répartir, on continue à partir du début du tableau, et ainsi de suite.
4. Écrire une fonction `repartirSauf` identique à `repartirTour`, sauf qu'elle ne repose jamais de graine dans la case qui a été vidée au début (le cas échéant, elle passe à la suivante).
5. Écrire une fonction `repartir` qui, après avoir fait la même chose que `repartirSauf`, examine le nombre de graines contenues dans la dernière case où une graine a été posée : si ce nombre est égal à `2` ou `3`, la case est vidée et la fonction renvoie ce nombre de graines (les graines ne sont pas réparties), sinon, la fonction renvoie `0`.
6. Écrire une fonction `jouer` qui prend en argument un tableau d'entiers, demande à l'utilisateur de lui donner l'indice d'une case, vérifie que cet indice est valable et que la case correspondante contient au moins une graine, sinon demande à l'utilisateur de modifier son choix, jusqu'à ce qu'il donne un choix correct ; ensuite la fonction appelle `repartir` et renvoie le nombre renvoyé par cette fonction : ces graines retirées du jeu sont gagnées pour le joueur.
7. Écrire une fonction `jeu` qui met en œuvre une variante d'awalé conformément à l'exemple ci-contre.

```
$ java Awale 6 10
[ 4 2 3 2 4 5 ]
Joueur 0. Quelle case? 1
[ 4 0 4 0 4 5 ] Joueur 0: 3. Joueur 1: 0.
Joueur 1. Quelle case? 5
[ 5 1 5 1 5 0 ] Joueur 0: 3. Joueur 1: 0.
Joueur 0. Quelle case? 2
[ 6 0 0 2 6 1 ] Joueur 0: 5. Joueur 1: 0.
Joueur 1. Quelle case? 4
[ 7 1 1 3 0 0 ] Joueur 0: 5. Joueur 1: 3.
Joueur 0. Quelle case? 1
[ 7 0 0 3 0 0 ] Joueur 0: 7. Joueur 1: 3.
Joueur 1. Quelle case? 0
[ 0 2 0 4 1 1 ] Joueur 0: 7. Joueur 1: 5.
Joueur 0. Quelle case? 4
[ 0 2 0 4 0 0 ] Joueur 0: 9. Joueur 1: 5.
Joueur 1. Quelle case? 3
[ 1 0 0 0 1 1 ] Joueur 0: 9. Joueur 1: 8.
Joueur 0. Quelle case? 5
[ 0 0 0 0 1 0 ] Joueur 0: 11. Joueur 1: 8.
Le joueur 1 a perdu.
```