

DM1 MP2I

Ce devoir est à déposer sur Cahier de prépa avant le mercredi 30 septembre 23h45. Dans une archive `nom_prenom.zip` (par exemple `DUPONT_Jean.zip` – sans espace entre les mots–) vous placez trois fichiers `exo1.c`, `exo2.c` et `exo3.c` correspondant aux 3 exercices. Une fonction d’affichage est demandée dans chacun : il est tout à fait possible de faire des copiés-collés de la fonction du premier fichier dans les deux autres. Pour chaque fonction (hormis l’affichage) `myfunc`, vous devez écrire une fonction de tests `void test_myfunc()`.

La fonction `main` contient juste des appels aux fonctions de tests.

Exercice 1. Le *tri par sélection* consiste à construire progressivement une partie triée au début du tableau.

À chaque étape, on recherche le plus petit élément de la partie qui n’est pas encore triée, puis on l’échange avec le premier élément de cette partie.

Question 1.

Écrire une fonction `void afficher(double tab[], int n)` qui affiche les `n` éléments d’un tableau de `double`. Par exemple, l’appel portant sur le tableau `{3.0, 1.0, 5.0}` doit produire un affichage de la forme :

```
[3, 1, 5]
```

Question 2.

Écrire une fonction `int indice_min(double tab[], int debut, int n)`.

La fonction doit renvoyer l’indice d’un plus petit élément parmi les éléments dont les indices sont compris entre `debut` et `n - 1`.

Par exemple, pour le tableau :

```
1 {4.0, 2.0, 5.0, 1.0, 3.0}
```

l’appel `indice_min(tab, 2, 5)` renvoie 3, car le plus petit élément de la partie `{5.0, 1.0, 3.0}` est `tab[3]`.

Question 3.

Écrire une fonction `void tri_selection(double tab[], int n)` qui trie par ordre croissant un tableau de `n` éléments.

À l’étape d’indice `i` :

- les éléments d’indices strictement inférieurs à `i` sont déjà à leur place définitive ;
- on recherche un plus petit élément parmi les cases d’indices `i` à `n - 1` ;
- on échange cet élément avec `tab[i]`.

La fonction utilisera la fonction `indice_min`.

Exercice 2. Le *tri par insertion* consiste à construire progressivement une partie triée du tableau. On considère d’abord que le premier élément est trié. On insère ensuite le deuxième élément à sa bonne place parmi les deux premiers, puis le troisième parmi les trois premiers, et ainsi de suite.

Pour insérer une valeur dans une partie déjà triée, on décale vers la droite tous les éléments qui lui sont strictement supérieurs, puis on place la valeur dans l’espace ainsi libéré.

Question 4.

Écrire une fonction `void afficher(double tab[], int n)` qui affiche les `n` éléments d’un tableau de `double`. Par exemple, l’appel portant sur le tableau `{1.0, 3.0, 5.0, 6.0}` doit produire un affichage de la forme :

```
[1.000000, 3.000000, 5.000000, 6.000000]
```

Question 5.

Écrire une fonction `int inserer(double tab[], int j)`.

Le tableau `tab` est de taille au moins `j + 1` et ses éléments d'indices compris entre 0 et `j - 1` sont déjà triés par ordre croissant.

La fonction doit insérer la valeur initialement située à l'indice `j` à sa bonne place parmi les éléments qui la précèdent. Elle doit pour cela :

- sauvegarder dans une variable `v` la valeur `tab[j]` ;
- décaler d'une case vers la droite les éléments supérieurs ou égaux à `v` ;
- placer `v` dans la case ainsi libérée ;
- renvoyer l'indice auquel `v` a été placée.

Pendant les décalages, une même valeur apparaît momentanément dans deux cases consécutives : le tableau contient toujours toutes les données utiles, mais l'une d'elles est temporairement dupliquée.

Par exemple, considérons le tableau :

```
1 {1.0, 3.0, 5.0, 6.0, 3.0}
```

Ses quatre premiers éléments sont triés et la valeur à insérer est `tab[4]`, c'est-à-dire 3.0. Après l'appel `inserer(tab, 4)`, le tableau devient :

```
1 {1.0, 3.0, 3.0, 5.0, 6.0}
```

La fonction renvoie alors 1, l'indice auquel la nouvelle occurrence de 3.0 a été placée.

Question 6.

Écrire une fonction `void tri_insertion(double tab[], int n)` qui trie par ordre croissant un tableau de `n` éléments.

La fonction utilisera la fonction `inserer`. À l'étape d'indice `j`, les éléments d'indices 0 à `j - 1` sont déjà triés et l'on insère `tab[j]` parmi eux.

Exercice 3. Le *tri à bulles* consiste à parcourir plusieurs fois le tableau en comparant deux éléments consécutifs. Lorsque ces deux éléments ne sont pas dans le bon ordre, on les échange.

À l'issue d'un premier parcours, le plus grand élément du tableau se trouve nécessairement dans la dernière case. Au parcours suivant, il suffit donc de s'arrêter à l'avant-dernière case, puis à l'antépénultième, et ainsi de suite.

Question 7.

Écrire une fonction `void afficher(double tab[], int n)` qui affiche les `n` éléments d'un tableau de `double`.

Par exemple, l'appel portant sur le tableau {3.0, 1.0, 5.0} doit produire un affichage de la forme :

```
[3, 1, 5]
```

Question 8.

Écrire une fonction `int une_passe(double tab[], int n)` qui parcourt les `n` premières cases du tableau.

Pour chaque indice `i` compris entre 0 et `n - 2`, la fonction compare `tab[i]` et `tab[i + 1]` et les échange lorsqu'elles ne sont pas dans l'ordre croissant.

La fonction renvoie le nombre d'échanges effectués.

Par exemple, à partir du tableau :

```
1 {4.0, 2.0, 5.0, 1.0, 3.0}
```

les transformations successives sont :

```
1 {2.0, 4.0, 5.0, 1.0, 3.0}
```

```
2 {2.0, 4.0, 1.0, 5.0, 3.0}
```

```
3 {2.0, 4.0, 1.0, 3.0, 5.0}
```

La fonction effectue trois échanges, renvoie donc 3 et le plus grand élément se trouve finalement dans la dernière case.

Question 9.

Écrire une fonction `void tri_bulles(double tab[], int n)` qui trie par ordre croissant un tableau de `n` éléments.

La fonction utilisera la fonction `une_passe`. Après chaque parcours, elle tiendra compte du fait que la partie finale du tableau est déjà triée et n'a plus besoin d'être parcourue.

Le tri pourra également s'arrêter avant d'avoir effectué tous les parcours si un appel à `une_passe` ne réalise aucun échange. Expliquer pourquoi le tableau est alors entièrement trié.