TP 2

Langage C – Tableaux

Compilation

On compilera avec la ligne de commande :

```
gcc -Wall -Wextra -Wvla -Werror -o nom_de_l_executable nom_du_fichier.c
```

Pour éviter de retaper cette commande à chaque compilation, pensez à utiliser les flèches haut et bas du clavier pour naviguer dans votre historique de commandes dans le terminal !

Récupérer les fichiers

Il y a deux fichiers à récupérer pour ce TP. Pour cela, allez sur la page :

https://informatique.rebout.fr

puis naviguez dans les répertoires jusqu'au TP du jour. Vous pouvez alors télécharger les fichiers :

- `TP02_C_Tableaux.c` qui est un « squelette » que vous allez completer ;
- `TP02_tests.h` qui contient une série de tests pour vérifier votre code.

Ces deux fichiers sont à enregistrer **dans le même répertoire**.

Tester les fonctions

Ce TP vient donc avec un jeu de tests. Par exemple, après avoir fini le premier exercice, compilez votre code et lancez le programme. Si tout se passe bien, vous devriez voir s'afficher cela :

```
Pour la fonction indice_max :
    Test numéro 1 : ok
    Test numéro 2 : ok
    Test numéro 3 : ok
```

Sinon, ça ressemblera plus à ça, qui indique que c'est le test numéro 2 qui n'est pas passé (vous pouvez aller chercher dans le fichier `TP02_tests.h` les détails de ce test pour comprendre ce qui n'a pas fonctionné dans votre code) :

```
Pour la fonction indice_max :
    Test numéro 1 : ok
    Test numéro 2 : essai: TP02_tests.h:38: test_indice_max: Assertion `
indice_max(t, 1) == 0' failed.
```

Si vous sautez un exercice, il suffit de commenter la ligne de test correspondante dans la fonction `main`.

1 Autour des maxima dans un tableau

Exercice 2.1 Écrire une fonction qui donne l'indice du maximum des éléments du tableau. Si ce maximum apparaît plusieurs fois, on renverra l'indice de la première occurence.

```
int indice_max(int tab[], int len)
```

Comme on l'a vu en cours, cette fonction prend deux arguments : le tableau à étudier, ainsi que sa taille.

On s'emploiera à écrire cette fonction avec une seule boucle `for` et sans faire appel à la fonction `maximum` écrite en cours.

Exercice 2.2 Écrire une fonction qui donne le nombre d'apparitions de l'élément maximum d'un tableau (ici aussi, avec un seul passage sur le tableau).

```
int nb_occurrences_max(int tab[], int len)
```

Pour les deux exercices suivants, on suppose que le tableau contient au moins deux éléments.

Exercice 2.3

Écrire une fonction qui donne le deuxième plus grand élément d'un tableau (toujours pareil, avec une seule boucle).

```
double deuxieme_plus_grand(double tab[], int len)
```

Exercice 2.4

1. Écrire une fonction qui prend en argument un tableau de double et sa longueur et renvoie la plus petite distance entre deux nombres de ce tableau.

   ```
   double plus_petite_distance(double tab[], int len)
   ```

2. Réécrire cette fonction, mais en supposant cette fois-ci que le tableau en entrée est trié par ordre croissant.

   ```
   double plus_petite_distance_trie(double tab[], int len)
   ```

Exercice 2.5 *Pour les rapides...*

Si on considère un tableaux de valeurs comparables (par exemple des nombres), on définit une case comme un maximum local si ses deux voisines ont des valeurs strictement inférieures à elle (une case avec une seule voisine n'est pas un maximum local).

1. Écrire une fonction qui prend en argument un tableau d'entiers, sa taille et un indice et teste si la valeur à cet indice est un maximum local.

   ```
   bool maximum_local(int t[], int len, int i)
   ```

2. Parfois, il y a des plateaux (plusieurs cases de suite qui ont la même valeur), et il faut donc aller chercher plus loin si la valeur descend ou non. Écrire une fonction en adaptant la fonction précédente pour que cela tienne compte de cette situation possible : une case est maintenant un maximum local si elle appartient à un plateau qui est encadré de valeurs strictement inférieures.

   ```
   bool vrai_maximum_local(int t[], int len, int i)
   ```

II Modifier un tableau

Exercice 2.6 On donne un tableau `t` de `double` de longueur `n`. On souhaite écrire une fonction qui va modifier le tableau pour qu'à la fin de l'exécution, la case d'indice `i` du tableau contienne la moyenne des éléments initiaux `t[0], t[1],...,t[i]`.

```
void moyenne_des_elements_precedents(double t[], int n)
```

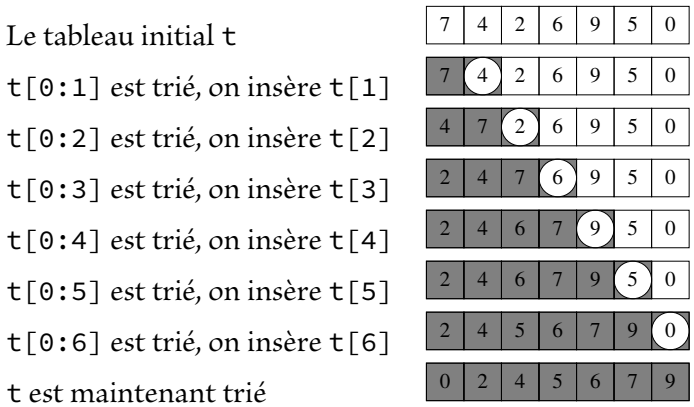
On rappelle que c'est l'adresse du tableau qui est passée à la fonction ; par conséquent, une commande du type `t[i] = ...` va modifier le tableau donné en argument en dehors de la fonction.

Exercice 2.7 **Tri par insertion.**

Le tri par insertion est l'un des plus naturels des algorithmes de tri. C'est souvent cette méthode que l'on suit sans y penser lorsqu'on trie ses cartes lors d'un jeu. Il consiste à insérer successivement chaque élément `a[i]` dans la portion du tableau `a[0:i]` déjà triée (j'utilise ici la notation Python bien pratique pour désigner une partie du tableau : `t[a:b]` correspond au sous-tableau de `t` constitué des éléments d'indice compris entre a inclus et b exclu) : on réalise cette étape en faisant « glisser » `a[i]` vers la gauche et en s'arrêtant dès que :

- on est en première position ;
- ou l'élément à sa gauche lui est inférieur ou égal.

Illustrons cette idée sur le tableau suivant :



1. Écrire une fonction qui échange les deux éléments d'indice `i` et `j` d'un tableau `t` (on suppose ici que les indices ne dépassent pas les bornes du tableau).

   ```
   void echange(int t[], int i, int j)
   ```

2. Écrire une fonction qui prend en arguments un tableau d'entiers `t`, de longueur n, et un entier `i`, en présupposant que :
   - $i \in [\![0, n-1]\!]$,
   - le sous-tableau `t[0:n]` est trié par ordre croissant ;

   et qui « déplace » l'élément `t[i]` afin que le sous-tableau `t[0:i+1]` soit trié.

   ```
   void insertion(int t[], int i)
   ```

3. Écrire une fonction qui réalise le tri par insertion.

   ```
   void tri_insertion(int t[], int len)
   ```

4. Tester votre fonction sur des exemples simple (on pourra pour cela écrire une petite fonction permettant d'afficher tous les éléments d'un tableau à la suite, séparés par un espace).

III Chaînes de caractères

Exercice 2.8 Écrire une fonction qui calcule la longueur d'une chaîne de caractères.

```
int longueur_chaine(char s[])
```

Cette fonction existe déjà dans la bibliothèque `string.h` sous le nom `strlen`. Bien entendu, ici, il faut s'en passer pour recoder la fonction.

Exercice 2.9 Écrire une fonction qui calcule le nombre d'occurences d'un caractère `c` dans une chaîne `s`.

```
int nb_occurrences(char c, char s[])
```

Exercice 2.10 Écrire une fonction qui :

- prend en entrée deux chaines `cible` et `source` **de même longueur** (on ne demande pas de le vérifier),
- et copie la chaine `source` dans la chaine `cible`.

```
void copie_chaine(char cible[], char source[])
```

Exercice 2.11 Écrire une fonction qui prend en entrée une chaîne de caractères `s` et qui la modifie pour retirer les lettres miniscules `p`, `a`, `r`, `c`.

```
void retirer_parc(char s[])
```

Remarquez que la chaîne obtenue sera sûrement plus courte, mais que le tableau en mémoire qui la contient n'aura pas changé de longueur. Il est donc fort probable qu'il y aura un caractère nul `\0` à placer quelque part.

Exercice 2.12 *Pour les rapides...*

Si vous ne connaissez pas, allez lire la page Wikipedia sur *ROT13*; ensuite écrire une fonction réalisant ce codage (on pourra se limiter en entrée à un texte non accentué écrit en miniscules).

```
void rot13(char s[])
```