1 Introduction

La recherche d'une donnée dans un ensemble d'informations volumineux est grandement facilitée lorsque ces informations sont *ordonnées*. Ainsi, si les mots d'un dictionnaire n'étaient pas ordonnés selon l'ordre alphabétique, il faudrait parcourir en moyenne la moitié du dictionnaire pour trouver un mot ! Le tri d'un ensemble de données est l'opération permettant de ré-arranger ces données selon un ordre choisi.

On se propose dans ce TP de trier un tableau de n entiers selon l'ordre croissant : à la fin du tri, les n entiers sont rangés du plus petit au plus grand dans les n premières cases du tableau. Le contenu du tableau à trier sera tiré aléatoirement par le programme. Plusieurs algorithmes de tri seront utilisés dans ce TP : le *tri par sélection* et plusieurs variantes du *tri à bulles*.

2 Balbutiements

2.1 Un premier programme

Dans un fichier **tri.c**, implémentez un programme qui effectue les actions suivantes :

1. demande à l'utilisateur de la saisie de n. Il faut que n soit être dans [1,30].
2. création d'un tableau **tab** qui contient n entiers,
3. initialisation de toutes les cases du tableau à 0,
4. affichage du contenu du tableau **tab**.

2.2 La fonction affiche_tableau

Afficher le contenu d'un tableau est une opération que l'on va faire plusieurs fois dans ce TP. Ainsi, il serait plus pratique de créer une fonction dont le rôle est uniquement d'afficher le contenu du tableau. Voici le prototype que peut avoir une fonction chargée de réaliser cet affichage :

```
void affiche_tableau(int taille, const int tableau[taille]);
```

Le paramètre **taille** représente la taille du tableau à afficher. Le paramètre **tableau** est le tableau à afficher. Le mot-clé **const** vous interdit de modifier le contenu du tableau, ce qui est mieux ici puisqu'un affichage n'est pas censé modifier ce qu'il affiche.

La manière dont le tableau doit être affiché est imposée. Par exemple, le tableau [42,37,51,88,4] doit être affiché de cette manière :

```
[42, 37, 51, 88, 4]
```

De plus, un caractère de fin de ligne '\n' doit être placé immédiatement après le crochet fermant.

Réalisez les opérations suivantes :
- implémentez la fonction **affiche_tableau**,
- modifiez votre fonction **main** pour que l'affichage n'y soit pas fait directement mais via un appel de la fonction **affiche_tableau**. Vous pouvez donner **n** et **tab** comme paramètres lors de l'appel de la fonction.

2.3 Remplissage aléatoire du tableau

On s'intéresse désormais au problème de remplissage d'un tableau par des valeurs aléatoires. Afin de générer des entiers aléatoires, vous pouvez vous servir de la bibliothèque *hasard* (au DLST) ou de la bibliothèque standard.

Le fonctionnement de la bibliothèque *hasard* est écrit dans votre polycopié. Si vous souhaitez vous servir de la bibliothèque standard, incluez les fichiers *stdlib.h* et *time.h* puis appelez au début du main le code suivant :

```
srand(time(0));
```

Ce code permet d'initialiser la graine aléatoire, afin d'éviter que les nombres générés soient toujours les mêmes. Ensuite, vous pouvez obtenir un nombre aléatoire entre 0 et RAND_MAX grâce à un appel de la fonction rand (man 3 rand).

Réalisez les opérations suivantes :
- Modifiez votre fonction **main** pour qu'elle initialise le tableau **tab** avec des entiers aléatoires entre 1 et 100, au lieu de tout initialiser à 0.

3 Tri par sélection

Dans cette partie du TP, nous allons travailler sur le fichier **tri_selection.c**. Pour commencer, copiez votre fichier **tri.c** en **tri_selection.c**.

Le tri par sélection est un algorithme de tri par comparaison. Cet algorithme n'est pas efficace puisqu'il demande de faire $\mathcal{O}(n^2)$ comparaisons pour trier n éléments. Cependant, il est important de connaître au moins un algorithme de tri, et la simplicité de cet algorithme en fait un bon candidat.

L'algorithme consiste à parcourir le tableau de gauche à droite, en plaçant un élément à sa place définitive à chaque parcours. L'algorithme peut se résumer ainsi :

```
Pour i de 0 à n-2 :
    i_min = indice_du_minimum(tab, i, n-1)
    permuter(tab, i, i_min)
Fin pour.
```

Un exemple d'exécution du tri par sélection sur le tableau [5, 1, 4, 2, 8] se trouve ci-dessous :

```
Étape 0 : [5, 1, 4, 2, 8]
Étape 1 : le minimum du tableau entre i=0 et i=4 a pour valeur 1
          et est à l'indice 1. On échange donc les cases 0 et 1.
          [1, 5, 4, 2, 8]
Étape 2 : le minimum du tableau entre i=1 et i=4 a pour valeur 2
          et est à l'indice 3. On échange donc les cases 1 et 3.
          [1, 2, 4, 5, 8]
Étape 3 : le minimum du tableau entre i=2 et i=4 a pour valeur 4
          et est à l'indice 2. Donc on ne fait rien (on échange
          les cases 2 et 2).
          [1, 2, 4, 5, 8]
Étape 4 : le minimum du tableau entre i=3 et i=4 a pour valeur 5
          et est à l'indice 3. Donc on ne fait rien (on échange
          les cases 3 et 3). On sait que la dernière case est
          triée par cet algorithme, donc on s'arrête ici.
```

Quand on lit l'algorithme du tri par sélection, on distingue plusieurs opérations distinctes :

— la recherche d'un minimum dans une sous-partie d'un tableau,
— l'échange de valeurs entre deux cases d'un tableau.

Ces opérations étant intéressantes en elles-mêmes (utilsables dans d'autres contextes que le tri par sélection), il peut être intéressant de les implémenter dans des fonctions propres. Cela ne vous est cependant pas imposé.

Réalisez les opérations suivantes :
— implémentez un tri par sélection dans votre fichier **tri_selection.c**. Vous pouvez pour cela vous organiser comme vous le souhaitez au niveau des fonctions à créer.

4 Le tri à bulles

Le tri à bulles est un autre algorithme de tri par comparaison. Tout comme le tri par sélection, il fait partie des algorithmes de tri les plus souvent enseignés en algorithmique élémentaire, de par sa simplicité, mais est peu utilisé en pratique à cause de sa faible efficacité.

Dans cet exercice, nous allons implémenter deux variantes du tri à bulles : la version classique (*bubble sort*) et une version plus optimisée (*cocktail sort*).

4.1 Première variante : le *bubble sort*

Dans cette partie du TP, nous allons travailler sur le fichier **tri_bulles.c**. Pour commencer, copiez votre fichier **tri.c** en **tri_bulles.c**.

Cet algorithme de tri prend en entrée un tableau T de taille n. Pour cela, il parcourt le tableau et compare des couples d'éléments successifs dans le tableau. Losque deux valeurs successives ne sont pas dans l'ordre croissant, on échange ces valeurs. Après chaque parcours complet du tableau, l'algorithme recommence l'opération. L'algorithme s'arrête lorsque aucun échange n'a eu lieu lors d'un parcours complet du tableau. On peut appeler **étape** un parcours complet du tableau. Vous trouverez ci-dessous un exemple d'exécution du *bubble sort* sur le tableau [5,1,4,2,8].

```
Étape 0 : [5, 1, 4, 2, 8]
Étape 1 : [1, 5, 4, 2, 8] 5>1 donc on a inversé 1 et 5
          [1, 4, 5, 2, 8] 5>4 don on a inversé 5 et 4
          [1, 4, 2, 5, 8] 5>2 donc on a inversé 5 et 2
          [1, 4, 2, 5, 8] 5<=8 donc on n'a rien fait. Fin du parcours.

Étape 2 : [1, 4, 2, 5, 8] 1<=4 donc on n'a rien fait
          [1, 2, 4, 5, 8] 4>2 donc on a inversé 4 et 2.
          [1, 2, 4, 5, 8] 4<=5 donc on n'a rien fait
          [1, 2, 4, 5, 8] 5<=8 donc on n'a rien fait. Fin du parcours.

Étape 3 : [1, 2, 4, 5, 8] On ne fait aucun échange lors de ce parcours.
                          Une fois à la fin du parcours, on s'arrête
                          puisque aucun échange n'a été réalisé.
```

Implémentez cet algorithme de tri dans le fichier **tri_bulles.c**. Vous êtes pour cela libres quant aux fonctions à définir. Si vous souhaitez définir et utiliser des fonctions, je vous propose les suivantes :

— une fonction dont le rôle est de réaliser une étape du tri à bulles,
— une fonction qui trie un tableau grâce à un tri à bulles (en appellant la fonction précédente).

4.2 Deuxième variante : le *cocktail sort*

Dans cette partie du TP, nous allons travailler sur le fichier **tri_cocktail.c**. Pour commencer, copiez votre fichier **tri_bulles.c** en **tri_cocktail.c**.

Un des problèmes du *bubble sort* que vous venez d'implémenter est qu'il parcourt plusieurs fois des cases qui sont déjà triées. En effet, lorsqu'on effectue un parcours complet d'un tableau vers la droite en échangeant toute paire de valeur qui n'est pas dans le bon sens, on place le maximum du tableau tout à droite. Lors du second parcours, on est sûr que le second plus grand élément est également à sa place...

Un autre problème est qu'en faisant toujours des parcours dans le même sens, il sera très long de déplacer les petites valeurs au début du tableau. Le *cocktail sort* propose de changer le sens de parcours du tableau à chaque étape. De plus, cet algorithme est conscient qu'après chaque parcours une valeur est à sa position finale et ne doit plus être parcourue. Voici un exemple d'exécution du *cocktail sort* :

```
Étape 0 : [5, 1, 4, 2, 8]
Étape 1 : on parcourt vers la droite de la case 0 à la case 4
          [1, 5, 4, 2, 8] 5>1 donc on a inversé 1 et 5
          [1, 4, 5, 2, 8] 5>4 don on a inversé 5 et 4
          [1, 4, 2, 5, 8] 5>2 donc on a inversé 5 et 2
          [1, 4, 2, 5, 8] 5<=8 donc on n'a rien fait. Fin du parcours.
                          On sait que 8 est à sa position finale.
```

```
Étape 2 : on parcourt vers la gauche de la case 3 à la case 0
          [1, 4, 2, 5, 8] 2<=5 donc on n'a rien fait
          [1, 2, 4, 5, 8] 4>2 donc on a inversé 4 et 2
          [1, 2, 4, 5, 8] 1<=2 donc on n'a rien fait. Fin du parcours.
                          On sait que 1 est à sa position finale.

Étape 3 : on parcourt vers la droite de la case 1 à la case 3
          [1, 2, 4, 5, 8] On ne fait aucun échange lors de ce parcours.
                          Une fois à la fin du parcours, on s'arrête
                          puisque aucun échange n'a été réalisé.
```

Cet algorithme de tri a deux conditions d'arrêt : il s'arrête lorsqu'aucun échange n'a été effectué dans une étape ou lorsque l'intervalle à trier ne comporte plus qu'un élément (ou moins).

Implémentez un tri cocktail dans le fichier **tri_cocktail.c**. Vous êtes libres des fonctions à définir et à utiliser, mais je vous propose les suivantes :

— Une fonction chargée de faire une étape du tri cocktail vers la droite
— Une fonction chargée de faire une étape du tri socktail vers la gauche
— Une fonction chargée de trier un tableau grâce à un tri cocktail (en appelant les deux fonctions précédentes).

5 Bonus : le *tri stupide*

Les trois algorithmes de tri que nous venons de voir ne sont pas très efficaces puisqu'ils nécessitent $\mathcal{O}(n^2)$ comparaisons, où n est le nombre d'élements du tableau. On peut cependant faire pire ! Le tri stupide (ou *bogosort*, ou *random sort*...) est un algorithme de tri particulièrement peu efficace. Il nécessite $\mathcal{O}((n+1)!)$ comparaisons en moyenne. Son principe est le suivant : tant que le tableau n'est pas trié, on le mélange aléatoirement. Cet algorithme s'arrête lorsqu'on a eu la chance d'obtenir un tableau trié.

Implémentez un tri stupide dans un fichier **tri_stupide.c**. Vous êtes pour cela libres des fonctions à définir et utiliser, mais je vous propose les suivantes :

- Une fonction chargée de permuter aléatoirement les cases d'un tableau
- Une fonction chargée de détecter si un tableau est trié ou non
- Une fonction chargée de trier un tableau grâce à un tri stupide (en appellant les deux fonctions précédentes).

6 Bonus : les algorithmes efficaces

Nous n'avons pas parlé d'algorithme de tri efficace jusqu'à présent puisque leur fonctionnement est plus complexe et qu'ils sont plus difficiles à implémenter.

Si vous avez implémenté tous les algorithmes précédents, je vous invite à vous renseigner sur Quicksort, Merge sort et Heapsort (cliquer sur le nom de l'algorithme vous renverra sur la page Wikipedia correspondante), qui sont trois des tris efficaces les plus souvent utilisés en pratique. Si vous le souhaitez, vous pouvez implémenter un ou plusieurs de ces algorithmes dans un fichier **tri_efficace.c**.