Informatique *DM*
Programmation en C Ipesup

À la fin de ce DM, vous serez capable de :

- Comprendre le principe général du tri d'un tableau.
- Expliquer et implémenter trois algorithmes classiques :
  1. Le **tri par insertion**.
  2. Le **tri rapide (quicksort)**.
  3. Le **tri fusion (mergesort)**.
- Comparer leurs performances, leur stabilité et leur usage mémoire.

Tri d'un tableau

Trier un tableau signifie réorganiser ses éléments pour qu'ils soient rangés du plus petit au plus grand. Par exemple :

$$[5,\ 2,\ 9,\ 1] \quad \longrightarrow \quad [1,\ 2,\ 5,\ 9].$$

Qu'est-ce qu'un tri *stable* ?

Lorsqu'on étudie un algorithme de tri, une propriété importante est la **stabilité**.

Un tri est dit **stable** lorsque : si deux éléments ont la même valeur, l'algorithme conserve l'ordre dans lequel ils apparaissent dans le tableau original.
Autrement dit, parmi les éléments égaux, l'algorithme ne modifie pas l'ordre relatif initial.

Exemple Considérons des éléments représentés par des paires (clé, id) :

$$[(12,5),\ (7,2),\ (12,8),\ (9,4)].$$

Les deux éléments de clé 12 sont dans l'ordre $(12,5)$ puis $(12,8)$.
Après un tri par clé :

- un tri stable donnera :

$$[(7,2),\ (9,4),\ (12,5),\ (12,8)];$$

- un tri non stable pourrait donner :

$$[(7,2),\ (9,4),\ (12,8),\ (12,5)].$$

Pourquoi la stabilité est-elle utile ? Elle est essentielle lorsque l'on trie des structures avec plusieurs champs et que l'on souhaite préserver un ordre préexistant. Par exemple :

- trier une liste d'étudiants par note en conservant l'ordre d'inscription ;
- effectuer plusieurs tris successifs (tri multi-clé).

Exercice 1 : mise en place du projet C

Vous allez travailler avec plusieurs fichiers C :

— `sort.h` : déclarations des fonctions de tri.
— `sort.c` : implémentations des fonctions de tri et fonctions utilitaires.
— `main.c` : programme de test.

1.1 Fichier `sort.h`

Ecrire un fichier `sort.h` contenant au minimum les déclarations suivantes :

```
#ifndef SORT_H
#define SORT_H

void print_array(int *a, int n);
int is_sorted(int *a, int n);

void insertion_sort(int *a, int n);
void quicksort(int *a, int n);
void mergesort(int *a, int n);

#endif
```

Questions :

1. À quoi sert, selon vous, la directive `#ifndef SORT_H` / `#define SORT_H` ?
2. Pourquoi, selon vous, est-il utile de séparer déclarations (`.h`) et définitions (`.c`) ?

1.2 Fichier `sort.c`

Créer le fichier `sort.c` et y inclure `sort.h` (comme montré ci-dessous).
Vous allez y implémenter progressivement les fonctions suivantes.

```
// Fichier sort.c
#include <stdio.h>
#include <stdlib.h>
#include "sort.h"

void print_array(int *a, int n) {
    /* A completer */
}

int is_sorted(int *a, int n) {
    /* A completer */
}

void insertion_sort(int *a, int n) {
    /* A completer */
}

void quicksort(int *a, int n) {
    /* A completer */
```

```
}

void mergesort(int *a, int n) {
    /* A completer */
}
```

Questions :

1. Que doit afficher `print_array` pour un tableau de taille n ?
2. Que doit vérifier précisément `is_sorted` ?

1.3 Fichier `main.c`

À faire : écrire un programme de test minimal :

```c
// Fichier main.c
#include <stdio.h>
#include <stdlib.h>
#include "sort.h"

int main(void) {
    int a[] = {5, 2, 9, 1, 5, 6};
    int n = sizeof(a) / sizeof(a[0]);

    print_array(a, n);
    printf("Tableau trie' ? %s\n", is_sorted(a, n) ? "OUI" : "NON");

    // Demander a l'utilisateur de choisir un algorithme de tri a tester :
    // 1 -> insertion_sort(a, n);
    // 2 -> quicksort(a, n);
    // 3 -> mergesort(a, n);

    print_array(a, n);

    return 0;
}
```

Questions :

1. Expliquez l'expression `sizeof(a) / sizeof(a[0])`.
2. Quel est le rôle de la fonction `is_sorted` pendant vos tests ?

Tri par insertion

Imaginez que vous triiez des cartes dans vos mains : vous prenez les cartes une à une et les insérez à la bonne position parmi celles déjà triées.

Principe :

— On considère que la partie gauche du tableau (indices 0 à $i-1$) est déjà triée.
— On insère l'élément d'indice i à sa place dans cette partie triée.

Exemple (à faire à la main) :
Prenez le tableau $[5, 2, 9, 1]$ et simulez manuellement le tri par insertion :

1. Après traitement de l'indice 1 : .....
2. Après traitement de l'indice 2 : .....
3. Après traitement de l'indice 3 : .....

Exercice 2 : implémentation du tri par insertion

Consigne : compléter la fonction suivante dans `sort.c` :

```
void insertion_sort(int *a, int n) {
    /* A completer :
       - utiliser une variable key
       - deplacer vers la droite les elements plus grands que key
       - inserer key a la bonne position
    */
}
```

Questions :

1. Pourquoi la boucle externe commence-t-elle à l'indice 1 ?
2. Que se passe-t-il si le tableau est déjà trié ?
3. Que se passe-t-il si le tableau est trié dans l'ordre inverse ?
4. Le tri par insertion est-il **stable** ? Justifiez.

Exercice 3 : tests

1. Construisez plusieurs tableaux de test :
   — Tableau déjà trié.
   — Tableau trié à l'envers.
   — Tableau avec beaucoup de valeurs identiques.
   — Tableau aléatoire.
2. Pour chaque tableau :
   — Affichez le tableau avant/après.
   — Vérifiez `is_sorted`.
3. (Optionnel) Ajoutez des `printf` dans la boucle interne pour visualiser les déplacements.

1 Tri rapide (Quicksort)

Le tri rapide (*quicksort*) utilise la stratégie *diviser pour régner* :

1. On choisit un élément du tableau, appelé **pivot**.
2. On réorganise le tableau de sorte que :
   — tous les éléments plus petits (ou égaux) au pivot sont à gauche ;
   — tous les éléments plus grands sont à droite.
3. On applique récursivement la même idée à la partie gauche et à la partie droite.

Exemple (à faire à la main) : Pour le tableau $[8, 3, 5, 2, 7]$ et pivot 5 :

1. Proposez une répartition des éléments en « partie gauche | pivot | partie droite ».
2. Que devient la partie gauche après tri récursif ? La partie droite ?

Exercice 4 : fonction de partition (schéma de Lomuto)

On souhaite écrire une fonction `partition` qui :

— prend en entrée un tableau `a` et deux indices `low` et `high` tels que
   — a[low] = premier élément du segment à trier
   — a[high] = dernier élément du segment à trier
— choisit un pivot (par exemple `a[high]`) ;
— réorganise les éléments pour placer le pivot à sa place définitive ;
— retourne l'indice où se trouve le pivot après la réorganisation.

Consigne : dans `sort.c`, ajouter une fonction *statique* :

```
static int partition(int *a, int low, int high) {
    /* A completer :
       - choisir un pivot, par exemple a[high]
       - utiliser deux indices i et j
       - echanger des elements pour regrouper les valeurs <= pivot a gauche
       - placer le pivot a sa position finale
       - retourner l'indice du pivot
    */
}
```

Questions :

1. Pourquoi est-il pratique de choisir le pivot comme `a[high]` ?
2. Après l'appel à `partition`, que dire des éléments à gauche et à droite du pivot ?

Exercice 5 : fonction `quicksort`

Consigne : compléter l'implémentation suivante :

```
void quicksort_rec(int *a, int low, int high) {
    /* A completer :
       - condition d'arret: si le segment a une taille <= 1
       - appeler partition pour obtenir la position p du pivot
       - trier recursivement la partie gauche et la partie droite
    */
```

```
}

void quicksort(int *a, int n) {
    /* A completer :
       - appeler quicksort_rec sur le segment entier [0, n-1]
    */
}
```

Questions de réflexion :

1. Que se passe-t-il si le tableau est déjà trié et que vous choisissez toujours le dernier élément comme pivot ?
2. Dans ce cas, la profondeur de récursion est-elle faible ou grande ?
3. Le tri rapide est-il stable ? Expliquez pourquoi.

Exercice 6 : tests et variantes

1. Testez votre `quicksort` sur différents tableaux (trié, inversé, aléatoire, avec doublons).
2. Modifiez le choix du pivot (par exemple, pivot au milieu du segment) et observez les effets possibles.
3. Ajoutez des `printf` pour suivre les appels récursifs (intervalle `[low, high]`).

Tri fusion (Mergesort)

Le tri fusion (*mergesort*) repose aussi sur le principe *diviser pour régner* :

1. Diviser le tableau en deux moitiés.
2. Trier récursivement chaque moitié.
3. Fusionner les deux moitiés triées en un tableau unique trié.

Exemple (à faire à la main) : Pour le tableau $[4, 1, 3, 2]$:

1. Dessinez les étapes de la division jusqu'à atteindre des segments de taille 1.
2. Montrez ensuite comment les segments sont fusionnés.

Exercice 7 : fonction de fusion

On souhaite écrire une fonction `merge` qui fusionne deux sous-tableaux contigus déjà triés.

Consigne : dans `sort.c`, écrire une fonction *statique* :

```
static void merge(int *a, int left, int mid, int right, int *tmp) {
    /* A completer :
       - fusionner les elements de [left, mid) et [mid, right)
         dans le tableau temporaire tmp
       - recopier ensuite la partie fusionnee dans a
    */
}
```

Questions :

1. Pourquoi a-t-on besoin d'un tableau temporaire `tmp` ?
2. Que se passe-t-il si, pendant la fusion, les éléments de gauche et de droite sont égaux ? Comment choisir lequel copier en premier pour obtenir un tri stable ?

Exercice 8 : fonction récursive `mergesort`

Consigne : compléter l'implémentation suivante :

```
static void mergesort_rec(int *a, int left, int right, int *tmp) {
    /* A completer :
       - condition d'arret: segment de taille <= 1
       - calculer mid = (left + right) / 2
       - trier recursivement [left, mid) et [mid, right)
       - fusionner avec merge
    */
}

void mergesort(int *a, int n) {
    /* A completer :
       - allouer un tableau tmp de taille n
       - appeler mergesort_rec(a, 0, n, tmp)
       - liberer tmp
    */
}
```

Exercice 9 : tests

1. Testez `mergesort` sur les mêmes tableaux que pour `insertion_sort` et `quicksort`.
2. Construisez un tableau de structures (par exemple, `struct { int cle; int id; }`) et triez par `cle`. Vérifiez si les éléments avec la même clé conservent leur ordre relatif (stabilité).

Synthèse et comparaison

Tableau récapitulatif

Algorithme	**Idée**	**Complexité**	**Mémoire**	**Stable**
Insertion	Insertion progressive	$O(n^2)$	$O(1)$	.....
Quicksort	Division par pivot	$O(n \log n)$ moyen, $O(n^2)$ pire	$O(\log n)$	.....
Mergesort	Division + fusion	$O(n \log n)$	$O(n)$	.....

Questions finales

1. Dans quel contexte utiliseriez-vous le tri par insertion plutôt qu'un autre ?
2. Dans quelles situations la stabilité du tri est-elle importante ?
3. Quel algorithme choisiriez-vous pour :
   — un très grand tableau aléatoire d'entiers ?
   — un petit tableau presque trié ?
   — un tableau d'objets où l'ordre relatif des éléments égaux est important ?