

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 :

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

### 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 :

```

1 #ifndef SORT_H
2 #define SORT_H
3
4 void print_array(int *a, int n);
5 int is_sorted(int *a, int n);
6
7 void insertion_sort(int *a, int n);
8 void quicksort(int *a, int n);
9 void mergesort(int *a, int n);
10
11 #endif

```

Questions :

1. À quoi sert la directive `#ifndef SORT_H / #define SORT_H`?

.....

2. Pourquoi 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`.

Vous allez y implémenter progressivement les fonctions suivantes.

```

1 void print_array(int *a, int n){
2     /* A compléter */
3 }
4
5 int is_sorted(int *a, int n) {
6     /* A compléter */
7 }
8
9 void insertion_sort(int *a, int n) {
10    /* A compléter */
11 }
12
13 void quicksort(int *a, int n) {
14    /* A compléter */
15 }
16

```

```

17 void mergesort(int *a, int n) {
18     /* A completer */
19 }

```

### 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 :

```

1 // Fichier main.c
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include "sort.h"
5
6 int main(void) {
7     int a[] = {5, 2, 9, 1, 5, 6};
8     int n = sizeof(a) / sizeof(a[0]);
9
10    print_array(a, n);
11    printf("Tableau trie' ? %s\n", is_sorted(a, n) ? "OUI" : "NON");
12
13    // Demander a l'utilisateur de choisir un algorithme de tri a tester :
14    // 1 -> insertion_sort(a, n);
15    // 2 -> quicksort(a, n);
16    // 3 -> mergesort(a, n);
17
18    print_array(a, n);
19    printf("Tableau trie' ? %s\n", is_sorted(a, n) ? "OUI" : "NON");
20
21    return 0;
22 }

```

### 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` :

```

1 void insertion_sort(int *a, int n) {
2     /* A completer :
3        - utiliser une variable key
4        - déplacer vers la droite les elements plus grands que key
5        - inserer key a la bonne position
6     */
7 }
```

## 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 et vérifiez `is_sorted`.

**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.  
.....  
.....

## 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* :

```

1 static int partition(int *a, int low, int high) {
2     /* A completer :
3         - choisir un pivot, par exemple a[high]
4         - utiliser deux indices i et j
5         - echanger des elements pour regrouper les valeurs <= pivot a gauche
6         - placer le pivot a sa position finale
7         - retourner l'indice du pivot
8     */
9 }
```

**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 :

```

1 void quicksort_rec(int *a, int low, int high) {
2     /* A compléter :
3         - condition d'arrêt: si le segment a une taille <= 1
4         - appeler partition pour obtenir la position p du pivot
5         - trier récursivement la partie gauche et la partie droite
6     */
7 }
8
9 void quicksort(int *a, int n) {
10     /* A compléter :
11         - appeler quicksort_rec sur le segment entier [0, n-1]
12     */
13 }
```

## Exercice 6 : tests et variantes

- Construisez plusieurs tableaux de test :
  - Tableau déjà trié.
  - Tableau trié à l'envers.
  - Tableau avec beaucoup de valeurs identiques.
  - Tableau aléatoire.
- Pour chaque tableau :
  - Testez votre `quicksort` et affichez le tableau avant/après.
  - Vérifiez `is_sorted`.
  - Modifiez le choix du pivot (par exemple, pivot au milieu du segment) et observez les effets possibles.
  - Ajoutez des `printf` pour suivre les appels récursifs (intervalle `[low, high]`).

### Questions :

- Que se passe-t-il si le tableau est déjà trié et que vous choisissez toujours le dernier élément comme pivot ?  
 .....  
 .....
- Dans ce cas, la profondeur de récursion est-elle faible ou grande ?  
 .....  
 .....
- Que se passe-t-il si le tableau est trié à l'envers et que vous choisissez l'élément du milieu comme pivot ?  
 .....  
 .....
- Le tri rapide est-il stable ? Expliquez pourquoi.  
 .....  
 .....

## 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* :

```

1 static void merge(int *a, int left, int mid, int right, int *tmp) {
2     /* A completer :
3        - fusionner les elements de [left, mid) et [mid, right)
4        dans le tableau temporaire tmp
5        - recopier ensuite la partie fusionnee dans a
6     */
7 }
```

**Questions :**

1. Pourquoi on a 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 ?

.....  
 .....

3. Comment choisir lequel copier en premier pour obtenir un tri stable ?

.....

## Exercice 8 : fonction récursive mergesort

**Consigne :** compléter l'implémentation suivante :

```

1 static void mergesort_rec(int *a, int left, int right, int *tmp) {
2     /* A compléter :
3         - condition d'arrêt: segment de taille <= 1
4         - calculer mid = (left + right) / 2
5         - trier récursivement [left, mid) et [mid, right)
6         - fusionner avec merge
7     */
8 }
9
10 void mergesort(int *a, int n) {
11     /* A compléter :
12         - allouer un tableau tmp de taille n
13         - appeler mergesort_rec(a, 0, n, tmp)
14         - libérer tmp
15     */
16 }
```

## 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

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 où l'ordre relatif des éléments égaux est important ? .....



## Fichier sort.h

```
1 #ifndef SORT_H
2 #define SORT_H
3
4 void print_array(int *a, int n);
5 int is_sorted(int *a, int n);
6
7 void insertion_sort(int *a, int n);
8 void quicksort(int *a, int n);
9 void mergesort(int *a, int n);
10
11
12 .....
13
14 .....
15
16 .....
17
18 .....
19
20 .....
21
22 .....
23
24 .....
25
26 .....
27
28 .....
29
30 .....
31
32 .....
33
34 .....
35
36 .....
37
38 .....
39
40 .....
41
42 .....
43
44 .....
45
46 .....
47 #endif
```

## Fichier sort.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "sort.h"
4
5
6 // Tri d'un tableau
7 void print_array(int *a, int n) {
8
9     .....
10
11     .....
12
13     .....
14
15     .....
16
17     .....
18
19     .....
20
21     .....
22
23     .....
24
25     .....
26 }
27
28 int is_sorted(int *a, int n) {
29
30     .....
31
32     .....
33
34     .....
35
36     .....
37
38     .....
39
40     .....
41
42     .....
43
44     .....
45
46     .....
47
48     .....
49 }
50
```

```
51 // Tri par insertion
52 void insertion_sort(int *a, int n) {
53
54     .....
55
56     .....
57
58     .....
59
60     .....
61
62     .....
63
64     .....
65
66     .....
67
68     .....
69
70     .....
71
72     .....
73
74     .....
75
76     .....
77
78     .....
79
80     .....
81
82     .....
83
84     .....
85
86     .....
87
88     .....
89
90     .....
91
92     .....
93
94     .....
95
96     .....
97
98     .....
99
100    .....
101
102    .....
103 }
```

```
104 //Quicksort
105
106 static int partition(int *a, int low, int high) {
107
108     .....
109
110     .....
111
112     .....
113
114     .....
115
116     .....
117
118     .....
119
120     .....
121
122     .....
123
124     .....
125
126     .....
127
128     .....
129
130     .....
131
132     .....
133
134     .....
135
136     .....
137
138     .....
139
140     .....
141
142     .....
143
144     .....
145
146     .....
147
148     .....
149
150     .....
151
152     .....
153
154     .....
155 }
156
```

```
157 void quicksort_rec(int *a, int low, int high) {
158
159     .....
160
161     .....
162
163     .....
164
165     .....
166
167     .....
168
169     .....
170
171     .....
172
173     .....
174
175     .....
176
177     .....
178
179     .....
180
181     .....
182
183     .....
184
185     .....
186 }
187
188 void quicksort(int *a, int n) {
189
190     .....
191
192     .....
193
194     .....
195
196     .....
197
198     .....
199
200     .....
201
202     .....
203
204     .....
205
206     .....
207 }
208
209
```

```
210 //Mergesort
211
212 static void merge(int *a, int left, int mid, int right, int *tmp) {
213
214     .....
215
216     .....
217
218     .....
219
220     .....
221
222     .....
223
224     .....
225
226     .....
227
228     .....
229
230     .....
231
232     .....
233
234     .....
235
236     .....
237
238     .....
239
240     .....
241
242     .....
243
244     .....
245
246     .....
247
248     .....
249 }
250
251 static void mergesort_rec(int *a, int left, int right, int *tmp) {
252
253     .....
254
255     .....
256
257     .....
258
259     .....
260
261     .....
262
```

```
263 .....
264 .....
265 .....
266 .....
267 .....
268 .....
269 .....
270 .....
271 .....
272 .....
273 .....
274 }
275 .....
276 .....
277 void mergesort(int *a, int n) {
278 .....
279 .....
280 .....
281 .....
282 .....
283 .....
284 .....
285 .....
286 .....
287 .....
288 .....
289 .....
290 .....
291 .....
292 .....
293 .....
294 .....
295 .....
296 .....
297 .....
298 .....
299 .....
300 }
```

## Fichier main.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "sort.h"
4
5 int main(void) {
6     // Demander a l'utilisateur de choisir une typologie de tableaux:
7     // 't' -> deja trie;
8     // 'e' -> trie a l'envers;
9     // 'i' -> avec des valeurs identiques;
10    // 'a' -> aleatoire
11
12    .....
13
14    .....
15
16    .....
17    // Selon le choix de l'utilisateur, enregistrer dans a le tableaux
18    // correspondant et calculer sa longueur
19
20    .....
21
22    .....
23
24    .....
25
26    .....
27
28    .....
29
30    .....
31
32    .....
33
34    .....
35
36    .....
37
38    .....
39
40    .....
41
42    print_array(a, n);
43    printf("Tableau trie' ? %s\n", is_sorted(a, n) ? "OUI" : "NON");
44
45    // Demander a l'utilisateur de choisir un algorithme de tri a tester et
46    // appliquez-le ensuite:
47    // 1 -> insertion_sort(a, n);
48    // 2 -> quicksort(a, n);
49    // 3 -> mergesort(a, n);

```



```
49
50 .....
51
52 .....
53
54 .....
55
56 .....
57
58 .....
59
60 .....
61
62 .....
63
64 .....
65
66 .....
67
68 .....
69
70 .....
71
72 .....
73 print_array(a, n);
74 printf("Tableau trie' ? %s\n", is_sorted(a, n) ? "OUI" : "NON");
75
76 return 0;
77 }
```