TD/TP de Programmation – Planche 3
Trieuses d'entiers et trieuses génériques

Le Tri Sélection (Selection Sort) et le Tri Insertion (Insertion Sort) sont deux algorithmes simples pour trier un tableau sur place. Ce sont les deux méthodes que les humains utilisent naturellement pour classer des objets physiques (paquets de cartes, fiches, copies d'examen).

Ces deux méthodes partagent la caractéristique qu'à chaque étape du tri, le tableau se divise en une partie déja triée et une partie non triée. L'étape consiste alors à transférer une carte de la partie non-triée à la partie triée. Le tableau est entièrement trié lorsque la totalité des cartes ont été transférées. Les deux méthodes diffèrent dans la façon de selectionner dans la partie non-triée la carte à transférer et comment celle-ci vient s'insèrer dans la partie triée.

```
Selection Sort:

 55   33   22   66   77   99   11
-----------------------------------
 55   33   22   66   77  *99*  11 |
 55   33   22   66  *77*  11 | 99
 55   33   22  *66*  11 | 77   99
*55*  33   22   11 | 66   77   99
 11  *33*  22 | 55   66   77   99
 11  *22*| 33   55   66   77   99
 11 | 22   33   55   66   77   99
-----------------------------------
 11   22   33   55   66   77   99
```

```
Insertion Sort:

 66   11   99   33   44   22   55
-----------------------------------
 66 | 11   99   33   44   22   55
*11*  66 | 99   33   44   22   55
 11   66  *99*| 33   44   22   55
 11  *33*  66   99 | 44   22   55
 11   33  *44*  66   99 | 22   55
 11  *22*  33   44   66   99 | 55
 11   22   33   44  *55*  66   99 |
-----------------------------------
 11   22   33   44   55   66   99
```

Principe du Tri Sélection

Dans le Tri Sélection, la partie non-triée est à gauche dans le tableau et la partie triée est à droite. La partie triée possède en plus la propriété de dominer la partie non-triée : son minimum est supérieur à tous les éléments de la partie non-triée. À chaque étape, on selectionne un élément maximal dans la partie gauche et on l'insère dans la partie triée. Du fait de la propriété de domination évoquée ci-dessus, l'élément sélectionné devient le nouveau minimum de la partie droite, et il va donc s'insérer devant celle-ci, en échangeant sa place de l'élément de la dernière case de la partie gauche.

La trace de l'encadré de gauche montre l'éxecution du Tri Sélection sur un tableau de 7 entiers, en affichant l'état du tableau sur une ligne à chaque étape. La barre verticale indique la séparation entre la partie gauche (non-triée) et la partie droite (triée). Les étoiles encadrent le nombre maximal de la partie gauche qui est sélectionné avant sa permutation avec le nombre immédiatement à gauche de la barre.

Principe du Tri Insertion

Dans le Tri Insertion, la partie non-triée est à droite et la partie triée est à gauche. Contrairement au tri sélection, la partie triée ne domine pas la partie non-triée. On selectionne l'élément le plus à gauche dans la partie non-triée et on l'insère au bon endroit dans la partie triée. Cette insertion provoque un décalage des plusieurs éléments, le dernier prenant la place initiale de l'élément déplacé.

La trace de l'encadré de droite montre l'éxecution du Tri Insertion. Les étoiles encadrent le nombre sélectionné après son insertion dans la partie triée.

Trieuses d'entiers

On se donne la structure `IntSelectionSorter` pour représenter l'état interne d'une trieuse de tableau d'entiers lors d'une étape du Tri Selection. La trieuse est en train de trier le tableau de longueur `nb_values` pointé par `values`. Les `nb_unsorted` premiers éléments sont non-triés, tandis que les autres le sont et les dominent. La position `selected_index` désigne l'emplacement de l'élément maximal actuellement selectionné par la trieuse.

```
typedef struct IntSelectionSorter {
  int * values;
  int nb_values;
  int nb_unsorted;
  int selected_index;
} IntSelectionSorter;
```

```
typedef struct IntInsertionSorter {
  int * values;
  int nb_values;
  int nb_sorted;
  int insertion_index;
} IntInsertionSorter;
```

De même, on se donne la structure `IntInsertionSorter` pour représenter l'état interne d'une trieuse de tableau d'entiers lors d'une étape du Tri Insertion. La trieuse est en train de trier le tableau de longueur `nb_values` pointé par `values`. Les `nb_sorted` premiers éléments sont déjà triés, tandis que les autres ne le sont pas encore. La position `insertion_index` désigne l'emplacement de l'élément actuellement inséré dans la partie triée.

1 Trieuse d'entiers `IntSelectionSorter`

1.1 Initialisation et affichage de la trace

Écrire la fonction `IntSelectionSorter_Init()` qui initialise une trieuse (pointée par) `s` avec un tableau `values` de `nb_values` entiers à trier (aucun élément n'est séléctionné). Écrire la fonction `IntSelectionSorter_Print()` qui affiche l'état de la trieuse dans le fichier `file`. Il s'agit d'afficher le tableau `values` sur une ligne, avec l'élément sélectionné (s'il y en a un) entouré d'une paire d'étoile, et avec la partie non-triée séparée de la partie triée par une barre verticale.

Tester ces fonctions en affichant une trieuse sur `stdout` après initialisation avec un tableau arbitraire, et en la réaffichant après avoir changé manuellement ses champs `selected_index` et `nb_unsorted`.

```
void IntSelectionSorter_Init  (IntSelectionSorter * s, int values [], int nb_values);
void IntSelectionSorter_Print (IntSelectionSorter const * s, FILE * file);

int MainTest_IntSelectionSorter_1 (void)
int main (void) { return MainTest_IntSelectionSorter_1 (); }
```

```
 55   33   22   66   77   99   11 |
 55   33   22 | 66   77  *99*  11
```

1.2 Sélection de l'élément maximal

Écrire la fonction `IntSelectionSorter_SelectMax()` qui selectionne l'élément maximal de la partie non triée du tableau `s->values`. Tester cette fonction en affichant une trieuse avant et après une selection.

```
void IntSelectionSorter_SelectMax (IntSelectionSorter * s);

int MainTest_IntSelectionSorter_2 (void);
int main (void) { return MainTest_IntSelectionSorter_2 (); }
```

```
 55   33   22   66   77   99   11 |
 55   33   22   66   77  *99*  11 |
```

1.3 Échange de deux élements

Écrire la fonction `IntSelectionSorter_SwapValues()` qui échange les éléments `s->values[k1]` et `s->values[k2]`. Tester cette fonction en affichant une trieuse avant et après une opération d'échange entre deux éléments arbitraires, par exemple le premier et le dernier.

```
void IntSelectionSorter_SwapValues (IntSelectionSorter * s, int k1, int k2);
int MainTest_IntSelectionSorter_3 (void);
int main (void) { return MainTest_IntSelectionSorter_3 (); }
```

```
 55   33   22   66   77   99   11 |
 11   33   22   66   77   99   55 |
```

1.4 Étape élémentaire du tri : la sélection suivie de l'échange

Écrire la fonction `IntSelectionSorter_DoSortStep()` qui effectue une étape élémentaire du Tri Selection sur le tableau `s->values`, à savoir la selection de l'élément maximal de la partie non-triée, suivi de son échange avec le dernier élément de la partie non triée, transférant ainsi cette case dans la partie triée. La fonction affiche la trace de son état dans le fichier `trace_file` si celui-ci n'est pas `NULL`. Tester cette fonction en effectuant quelques étapes sur une trieuse après initialisation, en traçant sur `stdout`.

```
void IntSelectionSorter_DoSortStep (IntSelectionSorter * s, FILE * trace_file);
int MainTest_IntSelectionSorter_4 (void);
int main (void) { return MainTest_IntSelectionSorter_4 (); }
```

```
 55   33   22   66   77  *99*  11 |
 55   33   22   66   77   11 | 99

 55   33   22   66  *77*  11 | 99
 55   33   22   66   11 | 77   99

 55   33   22  *66*  11 | 77   99
 55   33   22   11 | 66   77   99

*55*  33   22   11 | 66   77   99
 11   33   22 | 55   66   77   99
```

1.5 Tri Selection complet

En déduire la fonction `IntSelectionSorter_Sort()` qui effectue le tri complet du tableau `s->values` en traçant les étapes dans le fichier `trace_file` si celui-ci n'est pas `NULL`. Tester la fonction en affichant la trace du tri d'un tableau.

```
void IntSelectionSorter_Sort (IntSelectionSorter * s, FILE * trace_file);
int MainTest_IntSelectionSorter_5 (void);
int main (void) { return MainTest_IntSelectionSorter_5 (); }
```

```
 55   33   22   66   77  *99*  11 |
 55   33   22   66  *77*  11 | 99
 55   33   22  *66*  11 | 77   99
*55*  33   22   11 | 66   77   99
 11  *33*  22 | 55   66   77   99
 11  *22*| 33   55   66   77   99
 11 | 22   33   55   66   77   99
```

2 Trieuse d'entiers `IntInsertionSorter`

2.1 Initialisation et affichage de la trace

Écrire la fonction `IntInsertionSorter_Init()` qui initialise la trieuse (pointée par) `s`, avec un tableau `values` de `nb_values` entiers. La partie triée est réduite au premier élément et il n'y pas encore d'élément inséré. Écrire ensuite la fonction `IntInsertionSorter_Print()` qui affiche l'état interne de la trieuse dans le fichier `file`. L'élément inséré en dernier est affiché entouré d'une paire d'étoiles, et la partie triée est séparée de la partie non triée par une barre verticale.

Tester ces fonctions en affichant une trieuse sur `stdout` après initialisation avec un tableau arbitraire, et en la réaffichant après avoir changé manuellement ses champs `insertion_index` et `nb_unsorted`.

```
void IntInsertionSorter_Init  (IntInsertionSorter * s, int values [], int nb_values);
void IntInsertionSorter_Print (IntInsertionSorter const * s, FILE * file);

int MainTest_IntInsertionSorter_1 (void);
int main (void) { return MainTest_IntInsertionSorter_1 (); }
```

```
 66 | 11   99   33   44   22   55
 66   11  *99*  33 | 44   22   55
```

2.2 Étape élémentaire du tri : l'insertion d'un élément dans la partie triée

Écrire la fonction `IntInsertionSorter_Insert()` qui insère dans la partie triée de `s->values` le premier élément de sa partie non-triée. En déduire la fonction `IntInsertionSorter_DoSortStep()` qui effectue cette opération et en trace le résultat dans le fichier `trace_file` si celui-ci n'est pas `NULL`. Tester cette fonction en effectuant quelques étapes sur une trieuse après initialisation, en traçant sur `stdout`.

```
void IntInsertionSorter_Insert     (IntInsertionSorter * s);
void IntInsertionSorter_DoSortStep (IntInsertionSorter * s, FILE * trace_file);

int MainTest_IntInsertionSorter_2 (void);
int main (void) { return MainTest_IntInsertionSorter_2 (); }
```

```
 66 | 11   99   33   44   22   55
*11*  66 | 99   33   44   22   55
 11   66  *99*| 33   44   22   55
 11  *33*  66   99 | 44   22   55
```

2.3 Tri Insertion complet

En déduire la fonction `IntInsertionSorter_Sort()` qui effectue le tri complet du tableau `s->values` en traçant les étapes dans le fichier `trace_file` si celui-ci n'est pas `NULL`. Tester la fonction en affichant la trace du tri d'un tableau.

```
void IntInsertionSorter_Sort (IntInsertionSorter * s, FILE * trace_file);
int MainTest_IntInsertionSorter_3 (void);
int main (void) { return MainTest_IntInsertionSorter_3 (); }
```

Trieuses génériques

On souhaite pouvoir trier des données quelconques, et pas seulement des entiers, en utlisant les mêmes trieuses. Pour cela on se donne deux types de fonctions, `CompareFunc` et `PrintFunc` qui permettent respectivement de comparer deux données et d'afficher une donnée.

```
typedef int CompareFunc (void const * data1, void const * data2);
typedef void PrintFunc  (void const * data, FILE * file);
```

Les structures des trieuses vues précédemment sont modifiées de la façon suivante afin de devenir plus génériques : elles incluent la taille `cell_size` de chaque case en octets, ainsi que deux pointeurs sur fonctions `compare` et `print`, pour la comparaison et l'affichage des cases. Le tableau de `nb_cells` cases (et donc de `nb_cells * cell_size` octets) est pointé par `bytes`. Son type est `unsigned char *` plutôt que `void *` afin de faciliter l'arithmétique de pointeurs nécessaire à l'accès au cases.

```
typedef struct SelectionSorter {
  unsigned char * bytes;
  int nb_cells;
  int nb_unsorted;
  int selected_index;

  CompareFunc * compare;
  PrintFunc * print;
  size_t cell_size;
} SelectionSorter;
```

```
typedef struct InsertionSorter {
  unsigned char * bytes;
  int nb_cells;
  int nb_sorted;
  int insertion_index;

  CompareFunc * compare;
  PrintFunc * print;
  size_t cell_size;
} InsertionSorter;
```

3 Données génériques : exemples pour les entiers et les chaînes

Écrire les fonctions `CompareAsInt()` et `CompareAsString()` de type `CompareFunc`, qui permettent respectivement de comparer deux entiers et deux chaînes passés par pointeurs. Écrire les fonctions `PrintAsInt()` et `PrintAsString()` de type `PrintFunc`, qui permettent respectivement d'afficher un entier et une chaîne passés par pointeurs.

```
int CompareAsInt (void const * data1,
                  void const * data2);
void PrintAsInt  (void const * data
                  FILE * file);
```

```
int CompareAsString (void const * data1,
                     void const * data2);
void PrintAsString  (void const * data,
                     FILE * file);
```

```
int main (void)
{
  int a= 666, b= 777;
  CompareFunc * compare= CompareAsInt;
  PrintFunc * print= PrintAsInt;

  int comp= compare (& a, & b);
  PrintAsInt (& a, stdout);
  fprintf (stdout, "␣%d␣", comp);
  print (& b, stdout);
  fprintf (stdout, "\n");
  return 0;
}
```

```
int main (void)
{
  char * a= "ACE", * b= "BAD";
  CompareFunc * compare= CompareAsString;
  PrintFunc * print= PrintAsString;

  int comp= compare (& a, & b);
  PrintAsString (& a, stdout);
  fprintf (stdout, "␣%d␣", comp);
  print (& b, stdout);
  fprintf (stdout, "\n");
  return 0;
}
```

Attention, dans le code ci-dessus : si `&a` et `&b` sont des `int*` dans le `main()` de gauche, ce sont en revanche des `char**` dans le `main()` de droite, et non des `char*`. Penser à cette indirection supplémentaire dans `CompareAsString()` et `PrintAsString()`.

4 Trieuse générique `SelectionSorter`

Reprendre les fonctions de la trieuse d'entiers `IntSelectionSorter` et les adapter afin d'obtenir une trieuse générique pour le Tri Selection.

```
void SelectionSorter_Init (SelectionSorter * s, void * cells, int nb_cells,
                           CompareFunc * compare, PrintFunc * print, int cell_size);

void SelectionSorter_Print       (SelectionSorter const * s, FILE * file);
void SelectionSorter_SelectMax   (SelectionSorter * s);
void SelectionSorter_SwapCellsAt (SelectionSorter * s, int k1, int k2);
void SelectionSorter_DoSortStep  (SelectionSorter * s, FILE * trace_file);
void SelectionSorter_Sort        (SelectionSorter * s, FILE * trace_file);
```

On trouvera avantage à programmer les fonctions auxilliaires suivantes : `SelectionSorter_CellAt()` qui retourne l'adresse d'une case en fonction de son index ; et `ArrayOfBytes_Swap()` qui échange deux données de `nb_bytes` octets pointées par `bytes1` et `bytes2`.

```
void * SelectionSorter_CellAt (SelectionSorter const * s, int index);
void   ArrayOfBytes_Swap (char bytes1 [], char bytes2 [], int nb_bytes);
```

La trieuse doit fonctionner pour les entiers et les chaînes :

```
int main (void)
{
  int array[]= {
    55, 33, 22, 66, 77, 99, 11
  };
  SelectionSorter sorter;
  SelectionSorter_Init (& sorter, array, 7,
                        CompareAsInt,
                        PrintAsInt,
                        sizeof * array);
  SelectionSorter_Sort (& sorter, stdout);
  return 0;
}
```

```
int main (void)
{
  char * array[]= {
    "DO","RE","MI","FA","SOL","LA","SI"
  };
  SelectionSorter sorter;
  SelectionSorter_Init (& sorter, array, 7,
                        CompareAsString,
                        PrintAsString,
                        sizeof * array);
  SelectionSorter_Sort (& sorter, stdout);
  return 0;
}
```

5 Trieuse générique `InsertionSorter`

De même, reprendre les fonctions de la trieuse d'entiers `IntInsertionSorter` et les adapter afin d'obtenir une trieuse générique pour le Tri Insertion.

```
void InsertionSorter_Init (InsertionSorter * s, void * cells, int nb_cells,
                           CompareFunc * compare, PrintFunc * print, int cell_size);

void InsertionSorter_Print      (InsertionSorter const * s, FILE * file);
void InsertionSorter_Insert     (InsertionSorter * s);
void InsertionSorter_DoSortStep (InsertionSorter * s, FILE * trace_file);
void InsertionSorter_Sort       (InsertionSorter * s, FILE * trace_file);
```

On trouvera avantage à programmer les fonctions auxilliaires suivantes : `InsertionSorter_CellAt()` qui retourne l'adresse d'une case en fonction de son index ; et `ArrayOfBytes_Copy()` qui copie une donnée de `nb_bytes` octets pointée par `source` dans une donnée pointée par `dest`.

```
void * InsertionSorter_CellAt (InsertionSorter const * s, int index);
void   ArrayOfBytes_CopyFrom (char dest [], char source [], int nb_bytes);
```