TP #4 — Tri #1

Nous aurons l'occasion au cours de l'année (par exemple demain) de constater l'intérêt de trier des données, notamment comme étape préalable d'autres algorithmes. Le but de ce TP est d'implémenter plusieurs algorithmes effectuant ce tri pour des données de type entier, et qui utilisent des approches et possèdent des caractéristiques variées. On rencontrera notamment des tris qui sont ou ne sont pas :

— *En place* : un tri est dit *en place* s'il modifie son entrée telle qu'elle est triée à la fin de son exécution, et qu'il n'a pas besoin d'espace de stockage supplémentaire de taille significative.
— *Stable* : un tri est dit *stable* si l'ordre relatif des éléments identiques (pour l'ordre utilisé dans le tri) est préservé dans le résultat trié. Cette propriété est plus souvent utile lorsque l'on trie des éléments seulement relativement à une « partie » de leur valeur.
— *Par comparaison* : un tri est dit *par comparaison* si l'algorithme calcule son résultat uniquement en comparant les éléments à trier deux à deux (ce qui n'est *pas* la même chose que comparer un élément à trier à une valeur connue, comme `0`). La plupart des algorithmes de tri sont par comparaison, mais ce n'est pas le cas de tous.

Pour tout ce TP, il est vivement conseillé de compiler votre programme avec l'option `-fsanitize=address`. (en plus des options habituelles).

Échauffement

1. Écrivez une fonction C de signature :
```
bool is_sorted(size_t an, int a[an])
```
qui prend en entrée un tableau a de an éléments `int` et renvoie `true` si ceux-ci sont triés pour l'ordre croissants, et `false` sinon.

2. Écrivez une fonction C de signature :
```
bool test_is_sorted(void)
```
qui teste votre fonction `is_sorted` sur quelques (au moins trois) cas représentatifs.

3. Testez.

4. Supposez que vous souhaitiez tester qu'une certaine fonction de tri est correcte. Pourquoi n'est il pas *suffisant* de vérifier (avec une fonction similaire à `is_sorted`) que son résultat est trié ? Comment pourriez-vous compléter cette approche ?

Tri « chèvres et moutons »

On souhaite résoudre le problème suivant : soit un pré unidimensionnel occupé par des chèvres et des moutons à des positions quelconques, on souhaite renvoyer un nouveau pré de la même taille contenant autant de chèvres et de moutons que le pré précédent, mais non mélangés : les chèvres doivent occuper le début du pré, et les moutons la fin.
On modélise ce problème en représentant les *chèvres* par des entiers `0`, les *moutons* par des entiers `1`, et le pré par un tableau sur des `int` valant uniquement `0` ou `1` et indiquant en chaque position s'il y a une chèvre (`1`) ou un mouton (`0`).

5. De quelle autre façon aurait-on pu représenter un pré ?
6. Écrivez une fonction C de signature :
   ```
   void sag_sort(size_t an, int a[an])
   ```
   qui prend en entrée un tableau *a* de *an* `0` ou `1` et qui *modifie* celui-ci afin qu'il soit trié et contienne autant de `0` et de `1` qu'initialement. Vous pouvez traiter librement (comme votre propre *undefined behaviour*) le cas où le tableau initial contient d'autres éléments que des `0` ou `1`.
7. Écrivez une fonction C de signature :
   ```
   bool partial_test_sag(unsigned num)
   ```
   qui teste partiellement votre fonction `sag_sort` en vérifiant *num* fois sur des tableaux (par exemple de longueur 1000) générés pseudo-aléatoirement qu'elle produit un résultat trié. Vous pouvez par exemple utiliser `rand` ou (si vous utilisez votre propre ordinateur) `arc4random_uniform` pour la génération d'aléa. En cas de besoin, allez consulter le polycopié de cours (Section 10.2) ou appelez moi.
8. Testez (au moins avec la fonction précédente).

Tri par sélection, tri par bulle

On souhaite maintenant implémenter deux algorithmes de tri qui suivent la même idée suivante : pour trier un tableau *a* de *an* éléments (par ordre croissant), il suffit de trouver un élément maximum dans le tableau, de l'écrire à sa fin (en l'échangeant éventuellement avec l'élément présent à la fin du tableau), puis de trier le préfixe de *a* ainsi modifié constitué de ses *an-1* premiers éléments. On répète ce processus jusqu'à atteindre un préfixe d'un seul élément, qui est forcément trié. Quand il y a plusieurs éléments maximaux (égaux) lors d'une itération, le choix de celui qui est déplacé en fin de tableau fait qu'on aura un tri stable ou non, qui dans tous les cas sera en place et par comparaison.
De façon équivalente, on peut chercher un *minimum*, l'écrire en première position, puis faire de même avec le suffixe des *an-1* derniers éléments, etc.
On commence par implémenter le *tri par sélection*, qui n'est pas stable mais « échange » peu d'éléments.

9. Écrivez une fonction C de signature :

   ```
   void select_sort(size_t an, int a[an])
   ```

   qui prend en entrée un tableau a de an éléments `int` et le modifie afin qu'il contienne les mêmes éléments triés pour l'ordre croissant. Pour cela, utilisez une boucle *externe* pour un indice i de $an-1$ à 1 (inclus) et une boucle *interne* qui cherche un élément maximum parmi les éléments de a d'indice entre 0 et i (inclus), et une fois trouvé échange cet élément avec celui en position i. (Vous pouvez alternativement utiliser une boucle externe croissante de 0 à $an-1$ et une boucle interne de $i+1$ à an et extraire à chaque fois le minimum).
10. Écrivez une fonction C de signature :

    ```
    bool partial_test_select(unsigned num)
    ```

    similaire à `partial_test_sag` qui teste partiellement votre fonction `select_sort`.
11. Testez.

On demande maintenant d'implémenter le *tri par bulle*, qui est stable mais peut « échanger » plus d'éléments que le tri par sélection.

12. Écrivez une fonction C de signature :

    ```
    void bubble_sort(size_t an, int a[an])
    ```

    qui prend en entrée un tableau a de an éléments `int` et le modifie afin qu'il contienne les mêmes éléments triés pour l'ordre croissant. Pour cela, utilisez une boucle *externe* pour un indice i de $an-1$ à 1 (inclus) et une boucle *interne* qui compare un élément à son successeur et s'échange avec ce dernier si ce dernier est *strictement* c'est important pour la stabilité plus *petit* que lui. (Vous pouvez alternativement utiliser une boucle externe croissante de 0 à $an-1$ et une boucle interne de 0 à $an-i-1$ qui compare chaque élément à son successeur)
13. Écrivez une fonction C de signature ;

    ```
    bool partial_test_bubble(unsigned num)
    ```

    similaire à `partial_test_sag` qui teste partiellement votre fonction `bubble_sort`.
14. Testez.
15. *Instrumentez* vos fonctions de tri `select_sort` et `bubble_sort` afin de constater leur différence. Vous pouvez par exemple afficher (à l'aide de `printf`) l'échange (éventuel) d'élément effectué ou l'état courant du tableau à chaque étape.