TP C n°1

Tri de bâtons

Ce sujet de TP est une version modifiée d'un TP C++ réalisé en ALGPR[1] à Centrale Nantes.

Le but de ce TP est de programmer des algorithmes de tris sur des tableaux (ou des liste chaînées si vous le voulez). Les algorithmes de tris réalisés seront, dans l'ordre : le tri par insertion (Section II), le tri bulle (Section III), le tri rapide (Section IV). Ce TP est *visuel* : des fonctions sont disponibles dans le code companion pour l'affichage pendant les différents tris. Plus d'information sur l'environnement de développement se trouvent en Section VI. En fin de TP, nous analyserons la *performance* des différents tris codés (Section V) : la différence entre un $\mathcal{O}(n^2)$ et un $\mathcal{O}(n \log n)$, pour un tableau de taille n. On ne s'intéressera, dans ce TP, qu'à la complexité en *temps*.

I. Introduction, Quelques fonctions utilitaires

Dans cette section, nous codons les différents types de données et fonctions utiles pour le reste TP.

Q1. Définir le type de données `stick` représentant un bâton. Un bâton a une longueur (que l'on nommera `length`) et une couleur (que l'on nommera `color`), que l'on représentera par des entiers.

Définir un type `stick_stack` représentant un tableau de bâtons. Il contiendra un tableau de bâtons (nommé `sticks`) sous forme de pointeur, et un entier `n` représentant sa longueur.

On placera ces types dans un fichier `stick.h`.

Q2. Définir une fonction `bool compare(stick, stick)` pour comparer deux bâtons en utilisant l'*ordre lexicographique* (noté $\preccurlyeq_\ell$). On placera cette fonction dans un fichier `utils.c`. On rappelle que pour deux bâtons $b_1 = (\ell_1, c_1)$ et $b_2 = (\ell_2, c_2)$, représentés comme deux couples longueur–couleur, l'ordre lexicographique $\preccurlyeq_\ell$ est défini comme :

$$b_1 \preccurlyeq_\ell b_2 \iff \ell_1 < \ell_2 \text{ ou } (\ell_1 = \ell_2 \text{ et } c_1 \leq c_2).$$

Par la suite, on représentera dans un fichier les données d'une liste de bâtons sous la forme suivante. Sur la première ligne est indiqué le nombre de bâtons. Sur les lignes suivantes représentent les données de chaque bâton : on indique en premier la colour puis la longueur, séparées par un espace. On en donne un exemple dans la figure ci-contre (Code 1).

```
5
3 2
4 1
6 2
2 10
2 5
```

Code 1. Exemple de données avec le format décrit

Dans le fichier exemple `sticks_data.txt`, les couleurs vont de 0 à 6, et les longueurs vont de 1 à 20. Ces bornes ne sont valables ***que*** dans le fichier exemple `sticks_data.txt`.

Q3. Écrire une fonction `stick_stack read_from_file(char*)` qui lit les données d'un fichier dont le nom est donné en argument, et renvoie une liste de bâtons avec les champs correctement remplis. On placera cette fonction dans le fichier `utils.c`.

[1] ALGPR : Algorithmie et Programmation

Q4. Écrire une fonction `void write_to_file(char*, stick_stack)` qui écrit les données des bâtons dans un fichier dont le nom est donné en argument. On placera cette fonction dans le fichier `utils.c`.

On pourra tester les fonctions des questions Q3 et Q4 en lisant le fichier `sticks_data.txt`, et en le réécrivant dans un autre fichier. On comparera les deux fichiers.

Indication C. Dans le fichier `display.c` (téléchargeable sur le site internet), on définit trois fonctions pour l'affichage des bâtons :

- `init`, qui est à appeler *une fois* au début du programme, avant le premier affichage ;
- `finish`, qui est à appeler une fois à la fin du programme ;
- `show`, qui est à appeler dès que l'on veut afficher une liste de bâtons à l'écran.

Un fichier *header* `display.h` est disponible avec les prototypes des différentes fonctions :

```c
void init();
void finish();
void show(stick_stack);
```

Q5. Écrire la fonction principale `main` dans un fichier `main.c`. On lira les données contenues dans `sticks_data.txt`, et on les affichera à l'écran en utilisant les fonctions d'affichage.

On sera amené, dans la suite de ce TP, à changer le contenu de cette fonction `main`.

Dans les sections suivantes (Section II, Section III, et Section IV), on implémentera différents algorithmes de tris, que l'on visualisera avec les fonctions d'affichage déjà définies. Les implémentations seront placées dans un fichier `sort.c`.

II. Tri par insertion

Pour $i \in [\![1, n-1]\!]$ **faire**
  $j \leftarrow 0$
  Tant que $j < i$ et $b_i \preccurlyeq_\ell b_j$ **faire**
    $j \leftarrow j + 1$
  Fin tant que
  Insérer(b, i, j)
Fin pour

Procédure Insérer(b, i, j)
  $c \leftarrow b_i$
  Pour $k \in [\![j-1, i]\!]$[2] **faire**
    $b_k \leftarrow b_{k-1}$
  Fin pour
  $b_j \leftarrow c$
Fin procédure

Algorithme 1. Tri par insertion

Le tri par insertion consiste à insérer chaque carte à la suite des cartes inférieurs. C'est le tri généralement utilisé pour trier une « main » dans un jeu de cartes. On représente l'algorithme de tri dans la figure ci-contre (Algorithme 1).

Dans l'Algorithme 1, et dans les autres algorithmes par la suite, on note n le nombre de bâtons, $b = (b_0, \dots, b_{n-1})$ les différents bâtons et $\preccurlyeq_\ell$ l'ordre lexicographique sur les bâtons (codé en Q2).

La procédure Insérer consiste à placer dans la liste b l'élément b_i à l'indice j et ce, sans changer l'ordre des autres éléments. On supposera que lors de l'appel à la procédure Insérer que l'indice j est inférieur ou égal à l'indice i. Pour cela, on décale tous les éléments intermédiaires à droite, et on replace convenablement b_i et b_j.

Q6. Implémenter l'Algorithme 1 en codant la fonction `void insertion_sort(stick_stack)`.

[2] Attention, la boucle se parcourt à l'envers !

On ne vérifiera pas que la fonction de tri implémentée en Q6 est correcte. Ceci sera fait en Q8.

Q7. Visualiser le tri. Pour cela, on modifiera la fonction principale pour appeler la fonction codée en Q 6. On appellera la fonction `show` dans la boucle **Pour** de la procédure Insérer.

Q8. La visualisation de Q7 confirme-t-elle le fonctionnement de l'implémentation réalisée en Q6 ?

Q9. Quelle est la complexité pire cas de cet algorithme ?

Dans les sections suivantes (Section III et Section IV), on aura un « cheminement » similaire : on implémente l'algorithme donné, en ajoutant les appels à la fonction `show`, puis on vérifiera que la visualisation correspond à l'algorithme décrit, et on donnera sa complexité pire cas.

III. Tri bulle

Pour $i \in [\![0, n-1]\!]$ **faire**
  Pour $j \in [\![0, n-2-i]\!]$ **faire**
    Si $b_j \not\leqslant_\ell b_{j+1}$ **alors**
      Échanger b_j et b_{j+1}
    Fin si
  Fin pour
Fin pour

Algorithme 2. Tri bulle

Le tri bulle est l'algorithme le plus simple à coder. On considère deux éléments adjacents d'un tableau. S'ils ne sont pas dans le bon ordre, on les inverse. On répète cette opération pour chaque paire d'éléments dans le tableau. On répète cette répétition n fois. Le tri bulle déplace l'élément maximal vers la fin du tableau.

Q10. Implémenter l'Algorithme 2 dans une fonction de signature `void bubble_sort(stick_stack)`. Ajouter les appels à la fonction `show`, et changer la fonction `main` pour utiliser ce tri.

Q11. Visualiser le tri. Correspond-t-il à l'algorithme décrit ?

Q12. Quelle est sa complexité pire cas ?

IV. Tri rapide

Dans le tri rapide, on fixe un pivot (un élément choisi au hasard dans le tableau ou, en l'occurrence, le dernier élément), on sépare le tableau initial en deux sous-tableaux, d'un côté les éléments plus petits (pour la relation $\preccurlyeq_\ell$), et de l'autre les éléments plus grands, et on place le pivot au milieu. On trie récursivement les deux sous-tableaux.

Dans l'Algorithme 3, on appellera initialement TriRapide avec $d = 0$ et $f = n - 1$. Ces variables correspondent respectivement à l'indice de début et de fin du tableau.

Q13. Implémenter l'Algorithme 3 dans une fonction de signature `void quick_sort(stick_stack)`. Ajouter les appels à la fonction `show`, et changer la fonction `main` pour utiliser ce tri.

Q14. Visualiser le tri. Correspond-t-il à l'algorithme décrit ?

Q15. Quelle est sa complexité pire cas ?

On pourra implémenter et visualiser d'autres algorithmes de tri, par la suite. Mais, on aimerai observer l'*impact* des complexités de chacun des algorithmes, au vu des réponses aux questions Q9, Q12 et Q15.

Procédure TriRapide(b, d, f)
- **Si** $d \geq f$ or $d < 0$ **alors**
  - **Renvoyer** rien
- **Fin si**
- $p \leftarrow$ Partition(b, d, f)
- TriRapide(b, d, $p - 1$)
- TriRapide(b, $p + 1$, f)

Fin procédure

Procédure Partition(b, d, f)
- pivot $\leftarrow b_f$
- $i \leftarrow d - 1$
- **Pour** $j \in [\![d, f - 1]\!]$ **faire**
  - **Si** $b_j \preccurlyeq_\ell$ pivot **alors**
    - $i \leftarrow i + 1$
    - Échanger b_i et b_j
  - **Fin si**
- **Fin pour**
- $i \leftarrow i + 1$
- Échanger b_i et b_f
- **Renvoyer** i

Fin procédure

Algorithme 3. Tri rapide

V. Comparaisons des algorithmes de tris

Dans cette section, on comparera les différents algorithmes de tris en terme de *temps*. On mesurera donc le temps d'exécution des algorithmes en fonction de n. En particulier, on comparera le temps pour $n = 1\,000$, $n = 10\,000$ et $n = 100\,000$.

Pour tester avec valeurs de n variables, nous devons avoir un tableau b de taille n variable, d'où la question Q16.

Q16. Écrire, dans le fichier `utils.c`, une fonction `random_generator` ayant pour signature `stick_stack random_generator(int)` qui reçoit en argument le nombre n de bâtons à générer. On utilisera l'expression `rand() % m` pour obtenir un nombre (pseudo)aléatoire dans l'intervalle $[\![0,$ `m` $- 1]\!]$. On n'oubliera pas d'ajouter l'appel à la fonction `srand` au début du `main` pour initialiser le générateur (pseudo)aléatoire : on ajoutera ainsi le code `srand(time(NULL))`. On prendra soin de commenter les appels aux fonctions `display` qui ont été parsemées dans le code.

Indication C. On peut mesurer le temps d'exécution d'instructions en utilisant le module `time.h`. Après l'avoir importer, on peut utiliser ce module comme :

```c
const long double cps = CLOCKS_PER_SEC;

clock_t start = clock();
// CODE À EXÉCUTER
clock_t end = clock();

printf("Temps d'execution : %Lf s\n", (end - start) / cps);
```

> Le temps affiché est en secondes.

Q17. Analyser la performance des algorithmes de tri codés précédemment.[3] On pourra utiliser un graphe, et comparer avec les complexité moyenne des algorithmes données en Tableau 1.

Complexité	Pire cas	Cas moyen	Meilleur cas[4]
Tri par insertion	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
Tri bulle	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
Tri rapide	$\mathcal{O}(n^2)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$

Tableau 1. Complexité des algorithmes de tri

VI. Environnement de développement

Ce TP utilise une libraire externe pour l'affichage dans la console (`ncurses`, en l'occurrence). Ce TP est donc à réaliser *de préférence* sur « Repl.it ». L'environnement « Repl.it » est déjà pré-configuré et les fichiers pré-importés. Pour exécuter votre code, vous n'avez qu'à exécuter les commandes suivantes :

```
make    # Compile le programme (tous les fichiers)
./main  # Ex1cute le programme
```

Lors des questions « visuelles » (*i.e.* avec un affichage graphique), le terminal devra être assez grand. Si les bâtons ne s'affichent pas correctement, on pourra essayer d'agrandir la taille du terminal.

« Repl.it » pré-configuré	https://replit.com/@hugos29/Soutien-TP-1-C-Template
« Repl.it » corrigé	https://replit.com/@hugos29/Soutien-TP-1-C-Corrige

[3] Dans le test de performance, on prendra soin de générer des données aléatoires avec la fonction de Q16 pour *chaque* tri, et *chaque* valeur de n. On prendra également soin de les `free` après utilisation.

[4] Dans le meilleur cas, la liste est déjà triée, et cette verification se fait en $\mathcal{O}(n)$. Dans les algorithmes donnés, on n'a pas réalisé cette verification.

— Compétences —

- ☐ Maîtriser la mémoire dynamique avec `malloc` / `free`
- ☐ Générer des nombres aléatoires avec `rand` / `srand`
- ☐ Mesurer les performances d'un programme avec `time.h`
- ☐ Manipuler des tableaux (dynamiques)
- ☐ Utiliser une librairie externe (en l'occurrence, `display.h`)
- ☐ Connaître les différents algorithmes de tri
- ☐ Implémenter différents tris de tableaux plus ou moins efficaces
- ☐ Déterminer la complexité d'algorithmes
- ☐ Maîtriser l'ordre lexicographique $\preccurlyeq_\ell$
- ☐ Manipuler des fichiers (lecture, écriture)