TP1 : Algorithmes de tri

Dans ce TP, nous verrons différentes manières de trier un tableau de valeurs dans l'ordre croissant, et nous analyserons leurs avantages et inconvénients. Nous aborderons dans un premier temps deux algorithmes très simples, qui sont le tri à bulle (BubbleSort) et le tri par insertion (InsertionSort), puis nous étudierons un algorithme plus évolué, le tri fusion (MergeSort).

Le but de ce TP est de vous faire manipuler des pointeurs et des tableaux.

Introduction

Rappels complexité temporelle d'un algorithme

Pour avoir une idée des performances d'un algorithme, on utilise la notion de complexité. Pour faire simple, la complexité répond à la question suivante : pour des données d'entrée de taille n, combien d'opérations l'algorithme devra-t-il effectuer ?

Pour connaître la complexité d'un algorithme, il suffit donc de compter le nombre d'opérations élémentaires à effectuer pour des données d'entrée de taille n. Pour rendre compte de la complexité, on utilise la notation de Landau O(.). Par exemple, on peut définir les complexités O(n) ou $O(n^2)$.

Il n'est pas intéressant de connaître la complexité d'un algorithme si la taille des données est petite (car dans ce cas, le temps de traitement est de toute façon très court). C'est pourquoi on cherche plutôt à connaître la complexité asymptotique, c'est-à-dire vers combien d'opérations l'algorithme tend avec des données très grandes. On ne considère donc que le terme de plus grand ordre ; et on omet également le facteur multiplicatif. Par exemple, si un algorithme a besoin de $3n^2 + 5n$ opérations pour des données de taille n, on dira qu'il a une complexité quadratique $O(n^2)$.

Quelques catégories de complexité typiques (du plus rapide au plus lent) :

- O(1) : complexité constante
- O(log(n)) : complexité logarithmique
- O(n) : complexité linéaire
- O(n log(n)) : complexité quasi-linéaire
- $O(n^2)$: complexité quadratique
- $O(n^a)$: complexité polynomiale (a>1)
- $O(a^n)$: complexité exponentielle (a>1)
- O(n!) : complexité factorielle

Comme on ne cherche qu'une information sur le comportement asymptotique de l'algorithme, il n'est pas nécessaire de compter précisément le nombre d'opérations nécessaires (d'autant que la complexité et une considération théorique, indépendante de l'implémentation). Dans ce TP, on se contentera de compter le nombre de comparaisons nécessaires.

Attention, s'il est possible de comparer des algorithmes de complexités différentes (O(log(n)) est meilleur que O(n)), ce n'est pas le cas pour des algorithmes de même complexité : deux algorithmes O(n) ne sont pas forcément équivalents en termes de performance. On sait juste qu'ils sont meilleurs que les algorithmes O(n log(n)) et au-delà.

Algorithmes de tri

En informatique, il est souvent utile de trier une série de valeurs. Cela peut permettre par exemple d'utiliser des algorithmes plus complexes, comme des algorithmes de recherche ; ou bien de rendre

des données plus lisibles pour l'utilisateur (par exemple lorsque vous voulez afficher vos photos dans l'ordre chronologique).

Parce qu'ils sont aussi importants, les algorithmes de tri ont fait l'objet de très nombreuses recherches. Il existe aujourd'hui une multitude d'algorithmes pour trier des valeurs. Une question se pose alors : quel algorithme utiliser ? Il n'existe pas de réponse absolue à cette question. Chaque algorithme de tri possède ses propres caractéristiques.

Parmi les critères de classification possibles, on peut retenir les suivants :

- La complexité temporelle. On distingue plusieurs cas : la complexité dans le pire cas, très importante par exemple dans les systèmes temps-réel où on veut s'assurer que le traitement se fait dans un temps donné ; la complexité moyenne, et enfin la complexité dans le meilleur des cas (tableau déjà trié), en général moins importante, mais qui peut avoir son utilité dans certains cas.
  A noter que les algorithmes de tri qui fonctionnent sur la base de comparaisons ne peuvent avoir une complexité moyenne meilleure que O(n log(n)).
- La complexité spatiale. Donne une idée de l'espace mémoire supplémentaire requis pour faire fonctionner l'algorithme. Lorsqu'un algorithme a une complexité spatiale de O(1), on dit qu'il est « en place », ce qui signifie que tous les traitements se font directement sur le tableau à trier (pas besoin de mémoire supplémentaire pour stocker temporairement des éléments du tableau).
- Stabilité. Un algorithme de tri est stable s'il garde l'ordre relatif des valeurs égales (s'il y a des valeurs égales, celles-ci restent dans l'ordre initial). Cette caractéristique n'est évidemment pas utile dans des tableaux 1D, mais serait importante si on voulait trier des structures avec différents champs.
- Adaptabilité. Un algorithme est adaptatif s'il peut traiter les tableaux presque triés plus rapidement.

Travail de préparation

Revoir cours sur les pointeurs et tableaux. Ecrire le pseudo-code de BubbleSort (fonctionnement de l'algorithme en partie 1). Ecrire le Makefile pour compiler le projet constitué des fichiers suivants : main.c, fonctions.c et fonctions.h.

Pour ce TP, vous devrez trier les tableaux suivants :

- Ref contient des éléments déjà triés.
- Tab1 contient les mêmes éléments dans un ordre aléatoire.
- Tab2 contient les mêmes éléments dans l'ordre décroissant.
- Tab3 contient les mêmes éléments déjà triés, sauf les valeurs 20 et 81 qui sont interverties.

Les fonctions « affiche » et « compare » déjà codées vous permettront respectivement d'afficher le contenu d'un tableau, et de comparer deux tableaux. Les prototypes sont les suivants :

```
void affiche(int* a, int n);
int compare(int* a, int* b, int n);
```

Avec a et b deux tableaux de dimension n.

Pour voir certains algorithmes de tri en action :

https://www.toptal.com/developers/sorting-algorithms

Partie 1 : BubbleSort

BubbleSort est un algorithme de tri très simple à appréhender. Son fonctionnement est le suivant :

On compare le premier élément du tableau avec le second. Si le premier est plus grand, alors on permute les deux éléments. Puis on recommence avec les éléments 2 et 3, puis 3 et 4, etc. On parcourt ainsi le tableau plusieurs fois, jusqu'à ce qu'il soit trié.

Exemple :

5	1	4	2	3	Comparison & swap
1	5	4	2	3	Comparison & swap
1	4	5	2	3	Comparison & swap
1	4	2	5	3	Comparison & swap

1	4	2	3	5	Comparison
1	4	2	3	5	Comparison & swap
1	2	4	3	5	Comparison & swap
1	2	3	4	5	Comparison

1	2	3	4	5	Comparison
1	2	3	4	5	Comparison
1	2	3	4	5	Comparison
1	2	3	4	5	Comparison

On remarque qu'à chaque itération, la plus grande valeur non déjà triée est emmenée jusqu'à sa place définitive. C'est pour cette raison que ce tri s'appelle le tri à bulle : les plus grandes valeurs remontent progressivement vers la fin du tableau, comme des bulles d'air dans un liquide.

Note : cette considération peut amener à une optimisation de l'algorithme.

1. Compilez le projet avec votre Makefile et exécutez. Le programme doit afficher le contenu de tab1.

2. Codez la fonction « swap » qui intervertit les valeurs pointées par deux adresses.
   Voici le prototype de cette fonction : `void swap(int* a, int* b);`

3. Codez l'algorithme BubbleSort. Votre fonction prendra en argument le tableau à trier, ainsi que sa taille ; et retournera le nombre de comparaisons effectuées.

4. Vous pouvez vérifier le résultat de votre tri en utilisant la fonction « compare » déjà codée. Utilisez le tableau ref en argument ; il contient les éléments déjà triés.

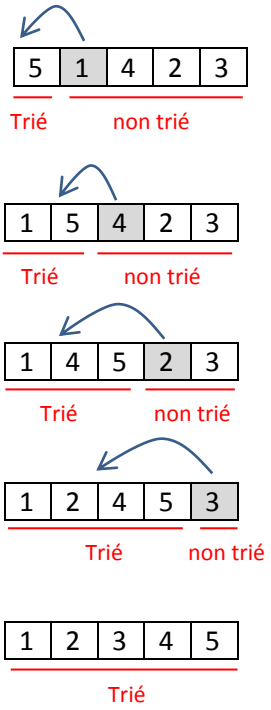
5. Testez votre algorithme sur les tableaux tab1, tab2, tab3 et ref. Relevez à chaque fois le nombre de comparaisons effectuées.

6. Quelle est la complexité temporelle de cet algorithme dans le meilleur des cas et dans le pire cas ? Est-ce un algorithme adaptatif ? Justifiez vos réponses.

7. Quelle est la complexité spatiale ? Est-ce un algorithme stable ?

Partie 2 : InsertionSort

Cet algorithme est également assez intuitif. On considère que les i-1 éléments du tableau sont triés. On prend ensuite l'élément i qu'on va insérer dans la partie déjà triée. On aura ainsi i éléments triés, et on pourra continuer avec l'élément i+1.



1. Codez la fonction insertionSort, qui prend en argument le tableau à trier, ainsi que sa taille ; et renvoie le nombre d'opérations effectuées.

2. Testez votre algorithme sur les tableaux tab1, tab2, tab3 et ref. Relevez à chaque fois le nombre de comparaisons effectuées.

3. Quelle est la complexité temporelle de cet algorithme dans le meilleur des cas et dans le pire cas ? Est-ce un algorithme adaptatif ? Justifiez vos réponses.

4. Quelle est la complexité spatiale ? Est-ce un algorithme stable ?

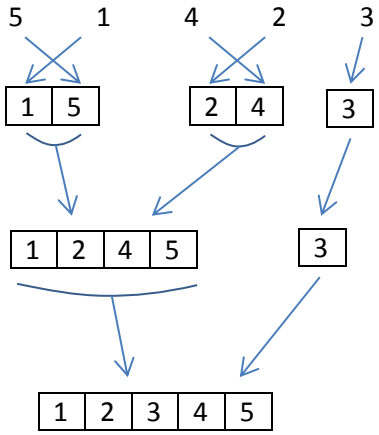
Même si InsertionSort a la même complexité asymptotique que BubbleSort, il est en réalité plus performant. InsertionSort est même considéré comme l'un des meilleurs algorithmes sur des entrées de petite taille, et sur des tableaux presque déjà triés.

Partie 3 : MergeSort

BubbleSort et InsertionSort ont tous deux une complexité moyenne de $O(n^2)$. Ils sont donc considérés comme des algorithmes « lents ». On leur préfère en général des algorithmes plus efficaces en termes de complexité temporelle, comme QuickSort (tri rapide), HeapSort (tri par tas) ou encore MergeSort (tri fusion). Nous allons maintenant nous intéresser à ce dernier cas.

MergeSort est un algorithme qui utilise la technique « diviser pour mieux régner » qui consiste à diviser le problème en plusieurs sous-problèmes faciles à résoudre, puis à assembler les solutions.

L'algorithme repose sur le fait qu'il est facile de combiner deux listes de valeurs déjà triées. On partira d'éléments isolés (une liste de une seule valeur est forcément déjà triée) qu'on combinera deux par deux. Puis on combinera ces doublets deux par deux pour donner des tableaux de 4 éléments, puis on combinera des tableaux de 4 éléments pour donner des tableaux de 8 éléments, etc.



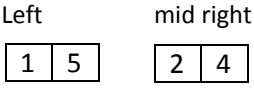
1. Pour réaliser l'algorithme, nous aurons besoin de créer un tableau temporaire (tmp) qui sera de même taille que le tableau à trier (tab). On sait que la taille du tableau est contenue dans la variable n.
   Que se passe-t-il si on déclare tmp de cette façon : « int* tmp = tab ; » ?
   Que se passe-t-il si on déclare tmp de cette façon : « int tmp[n]; » ?

2. Quelle est la manière correcte de créer ce tableau de taille n ? Cette technique devra être utilisée en question 4.

3. Codez la fonction « merge » qui permet de combiner les valeurs de deux tableaux.

   ```
   void merge (int* tab, int* tmp, int left, int mid, int right,
   int* cnt);
   ```

   tab est le tableau à trier.
   Tmp est un tableau (même taille que tab) qui permet de stocker de manière temporaire les éléments du tableau.
   Cnt est un pointeur vers un compteur du nombre d'opérations effectuées.
   Left est l'indice du premier élément du premier tableau à combiner.
   Mid est l'indice du premier élément du deuxième tableau à combiner.
   Right est l'indice du dernier élément du deuxième tableau à combiner.
   Exemple : pour la recombinaison de deux tableaux de 2 éléments :

4. Codez maintenant la fonction mergeSort qui fera appel à la fonction « merge » pour combiner itérativement des tableaux de plus en plus grands. (Attention, le tableau n'a pas un nombre d'éléments puissance de 2). Commencez par créer un tableau de taille n, qui sera utilisé par la fonction merge. *N'oubliez pas de libérer la mémoire à la fin.*
   Tout comme les deux premiers tris, la fonction prend en argument le tableau à trier et sa taille ; et renvoie le nombre d'opérations effectuées.

5. Testez votre algorithme sur les tableaux tab1, tab2, tab3 et ref. Relevez à chaque fois le nombre de comparaisons effectuées. Que remarquez-vous ?

6. Quelle est la complexité temporelle de cet algorithme ? Justifiez

7. Quelle est sa complexité spatiale ? Est-il stable ?

Bonus : pour aller plus loin : pointeurs de fonction

Tout comme les différents types de variables (int, char, double, etc), il est possible de créer des pointeurs vers des fonctions.

Ces pointeurs ont toutefois une signification différente des pointeurs de variables. Les pointeurs de fonction pointent vers du code et non vers des données.

Soit une fonction f qui a pour prototype :

```
void f(int a, char b);
```

Pour déclarer un pointeur de fonction qui pourra pointer vers f, il faut donner les informations sur le type de retour et le type des arguments. Ici, on crée un pointeur f_ptr qui pourra pointer vers n'importe quelle fonction qui prend en entrée un int et un char, et qui ne retourne rien (void) :

```
void (*f_ptr)(int, char);
```

Pour faire pointer f_ptr sur la fonction f, deux possibilités (on remarque que la syntaxe est plus laxiste que pour les pointeurs de données) :

```
f_ptr = &f;
f_ptr = f;
```

On peut ensuite utiliser le pointeur de fonction f_ptr pour exécuter la fonction f. Bien entendu, il ne faut pas oublier d'envoyer des valeurs aux arguments et de récupérer la valeur de retour si besoin. Ici encore, deux syntaxes possibles :

```
(*f_ptr)(8, 'a');
f_ptr(8, 'a');
```

Tout comme les pointeurs de variables, les pointeurs de fonction peuvent être utilisés comme argument d'une fonction (ou même comme valeur de retour).

1. Créez deux fonctions asc et desc qui prennent en argument deux valeurs et renvoient 0 ou 1 suivant si la première valeur est inférieure ou supérieure (respectivement).

2. Modifiez votre fonction bubbleSort pour qu'elle prenne en argument un pointeur vers une des deux fonctions asc ou desc.

3. Dans bubbleSort, remplacez le test par un appel via le pointeur de fonction. Ainsi, lorsque vous faites appel à bubbleSort, vous pouvez choisir si vous voulez un tri en ordre ascendant ou descendant. Et si vous voulez inventer un autre moyen de trier, il suffira de créer une nouvelle fonction de comparaison ; pas besoin de réécrire complètement bubbleSort.