II.1102 : Algorithmique et Programmation
TP 5 : Tris et Complexité algorithmique

Patrick Wang

20 novembre 2020

Table des matières

1 Objectifs du TP

— Montrer l'intérêt de concevoir des algorithmes optimisés ;
— Manipuler les algorithmes de tri vus en cours ;

2 Rappels

2.1 Création d'un tableau

Il y a deux possibilités lorsque l'on souhaite créer un tableau :
— Soit on connaît sa taille ;
— Soit on connaît tous ses éléments.
Il faut alors utiliser une des deux instructions suivantes :

```
int[] tableau = new int[100];
char[] lettres = {'a', 'b', 'c'};
```

2.2 Algorithmes de tri

Lors de la séance de cours, nous avons vu plusieurs algorithmes de tri : le tri par sélection, le tri par insertion, le tri à bulles, et le *quick sort*. Cette liste est bien évidemment non-exhaustive, et il existe un grand nombre d'algorithmes différents permettant de trier un tableau.

Il est justifié de se poser la question : pourquoi avoir autant d'algorithmes différents ? L'objectif de ce TP est de vous montrer que, sur des jeux de données de taille modérée, optimiser son code peut avoir de l'importance.

Cette optimisation peut concerner des algorithmes de tri (comme nous allons le voir lors de ce TP), mais peut être généralisé à tout algorithme traitant un grand jeu de données.

2.3 Complexité algorithmique

La complexité algorithmique est la mesure généralement utilisée pour déterminer les performances d'un algorithme. Comme expliqué en cours, la complexité algorithmique va compter le nombre d'instructions effectuées. La notation utilisée est la notation de Landau $\mathcal{O}$.

Puisque la complexité algorithmique compte le nombre d'instructions, on va souvent l'exprimer en fonction de la taille du jeu de données utilisé par l'algorithme. C'est pourquoi nous avons souvent des complexités en $\mathcal{O}(n)$, $\mathcal{O}(n^2)$, $\mathcal{O}(2^n)$, $\mathcal{O}(n!)$, ... La page Wikipédia portant sur la complexité algorithmique présente un certain nombre de *classes de complexité* qu'il peut être intéressant de connaître.

3 Exercices

3.1 Comparaison des algorithmes de tri

Dans cet exercice, nous allons créer un *grand* tableau d'entiers, que nous allons ensuite copier plusieurs fois afin de lui appliquer les algorithmes de tri vus en cours. Cela nous permettra de nous rendre compte qu'optimiser son programme peut être important.

Dans cet exercice, nous allons aussi calculer la durée de chaque fonction implémentée afin de voir les différences de durées. Pour ce faire, nous pourrons utiliser les instructions suivantes :

```
Instant start = Instant.now();
// Des instructions qui peuvent prendre du temps
Instant end = Instant.now();
// duration va donc contenir, en ms, la duree ecoulee entre end et start
long duration = Duration.between(start, end).toMillis();
```

3.1.1 Création d'un *grand* tableau d'entiers

En Java, il existe l'instruction `Integer.MAX_VALUE` permettant de récupérer la plus grande valeur qu'il est possible de stocker dans une variable de type `int`. Nous allons nous servir de cette valeur pour créer notre tableau d'entiers.

Questions :

— Déclarez une variable statique `SIZE` de type `int` qui va stocker la taille du tableau. Initialisez cette variable à `Integer.MAX_VALUE / 1000`;
— Déclarez un tableau d'entiers en tant que variable statique de classe et initialisez sa taille à `Integer.MAX_VALUE / 1000`. Cela initialisera un tableau d'entiers de taille 2,147,484, qui n'est finalement pas si grand que ça compte-tenu des problématiques de big data actuelles (pour rappel, le fichier `dept2018.csv` comptait un peu plus de 3 millions de lignes).

Remarques :

— Si votre programme plante à l'exécution avec une erreur mentionnant une allocation mémoire impossible, c'est que votre tableau est trop grand pour l'espace mémoire accordé à votre IDE. Réduisez le donc.
— Si, au cours du TP, votre programme met trop de temps à se terminer, on pourra encore plus réduire la taille du tableau par un facteur 10.

3.1.2 Initialisation du tableau avec des valeurs aléatoires

Nous allons initialiser le tableau avec des valeurs aléatoires comprises entre 0 et `Integer.MAX_INT`. De plus, nous allons chercher à déterminer la durée de cette étape d'initialisation. Pour cela, nous allons créer la fonction suivante :

```
public static void initialiserTableau() {
    Instant start = Instant.now();
    System.out.println("Debut d'initialisation...");
    Random random = new Random();
    for (int i = 0; i < tableau.length; i++) {
        tableau[i] = random.nextInt(SIZE);
    }
    Instant end = Instant.now();
    long duration = Duration.between(start, end).toMillis();
    System.out.println("L'initialisation a pris " + duration + " ms");
}
```

Appelez cette fonction dans le `main()`.

3.1.3 Création des copies du tableau initial

Nous allons créer autant de copies du tableau initial que l'on implémentera d'algorithmes de tri.

Java nous fournit une instruction nous permettant de créer un *clône* du tableau initial. Ce clône va contenir les mêmes valeurs mais se situera à une adresse mémoire différente. Cela a pour conséquence que les modifications effectuées sur le clône n'auront pas de répercussion sur le tableau principal. Ce détail est important dans notre cas, puisque nous souhaitons trier un même tableau avec plusieurs algorithmes différents.

Dans le `main()`, sous l'appel de la fonction `initialiserTableau()`, insérez les instructions suivantes :

```
int[] tableauSelection = new int[SIZE];

// arraycopy(src, startIndex, dest, startIndex, size)
System.arraycopy(tableau, 0, tableauSelection, 0, SIZE);
```

3.1.4 Tri par sélection

Questions :
- Quelle est la complexité algorithmique du tri par sélection ?
- Créez une fonction `triSelection(int[] tableau)`, créez une copie de `tableau` et implémentez le tri par sélection sur cette copie ;
- Ajoutez les instructions nécessaires pour calculer la durée écoulée lors du tri du tableau ;
- Appelez cette fonction dans le `main()` afin de déterminer la durée nécessaire pour trier un tableau d'environ 2 millions d'entrées. Si le tri dure trop longtemps, on pourra diviser la taille par 10.
- On pourra, si on le souhaite, créer une boucle pour répéter plusieurs fois ce tri et faire une moyenne sur le temps d'exécution.

3.1.5 Tri par insertion

Questions :
- Quelle est la complexité du tri par insertion ?
- Créez une fonction `triInsertion(int[] tableau)`, créez une copie du `tableau`, puis implémentez le tri par insertion sur cette copie.
- Ajoutez les instructions nécessaires pour calculer la durée écoulée lors du tri du tableau ;
- Appelez cette fonction dans le `main()` afin de déterminer la durée nécessaire pour trier le tableau d'environ 2 millions d'entrées. Si le tri dure trop longtemps, on pourra diviser la taille par 10.
- On pourra, si on le souhaite, créer une boucle pour répéter plusieurs fois ce tri et faire une moyenne sur le temps d'exécution.

3.1.6 Tri à bulles

Questions :
- Quelle est la complexité du tri à bulles ?
- Créez une fonction `triBulles(int[] tableau)`, créez une copie du `tableau`, puis implémentez le tri à bulles pour cette copie.
- Ajoutez les instructions nécessaires pour calculer la durée écoulée lors du tri du tableau ;
- Appelez cette fonction dans le `main()` afin de déterminer la durée nécessaire pour trier le tableau d'environ 2 millions d'entrées. Si le tri dure trop longtemps, on pourra diviser la taille par 10.
- On pourra, si on le souhaite, créer une boucle pour répéter plusieurs fois ce tri et faire une moyenne sur le temps d'exécution.

3.1.7 Quick sort

Questions :
- Quelle est la complexité du tri *quick sort* ?
- En vous aidant du cours, implémentez les deux fonctions :
  - `quicksort(int[] tableau, int indGauche, int indDroit)`
  - `partition(int[] tableau, int indGauche, int indDroit)`
- Ajoutez les instructions nécessaires pour calculer la durée écoulée lors du tri du tableau ;

— Appelez cette fonction dans le `main()` afin de déterminer la durée nécessaire pour trier le tableau d'environ 2 millions d'entrées. Si le tri dure trop longtemps, on pourra diviser la taille par 10.
— On pourra, si on le souhaite, créer une boucle pour répéter plusieurs fois ce tri et faire une moyenne sur le temps d'exécution.

3.1.8 Tri natif de Java

Java implémente un algorithme de tri natif intitulé le *double pivot quick sort*. Il s'utilise de la façon suivante :

```
int[] tableau = { ... };
Arrays.sort(tableau);
// A la fin de cette etape, tableau est trie
```

Questions :

— Créez une fonction `triJava(int[] tableau)`, créez une copie de `tableau`, et appelez la fonction `Arrays.sort()` sur cette copie ;
— Ajoutez les instructions nécessaires pour calculer la durée écoulée lors du tri du tableau ;
— Appelez cette fonction dans le `main()` afin de déterminer la durée nécessaire pour trier le tableau d'environ 2 millions d'entrées. Si le tri dure trop longtemps, on pourra diviser la taille par 10.
— On pourra, si on le souhaite, créer une boucle pour répéter plusieurs fois ce tri et faire une moyenne sur le temps d'exécution.