II.1102 : Algorithmique et Programmation
TP 3 : Tris et Complexité algorithmique

Patrick Wang

14 et 18 Octobre 2019

1 Objectifs du TP

— Continuer la manipulation de tableaux ;
— Montrer l'intérêt de concevoir des algorithmes optimisés ;
— Manipuler les algorithmes de tri vus en cours ;
— Retravailler le TP noté et voir un corrigé.

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_INT` 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 `SIZE` de type `int` qui va stocker la taille du tableau. Initialisez cette variable à `Integer.MAX_INT / 1000` ;
- Déclarez un tableau d'entiers en tant que variable de classe (c'est-à-dire, en dehors et au dessus de la fonction `main()`) et initialisez sa taille à `Integer.MAX_INT / 1000`. Cela initialisera un tableau de taille 2,147,484 que $\approx$ 2 millions d'entrées ne représente pas grand chose désormais.

Remarque :

- 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)` et implémentez le tri par sélection ;
- 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.

3.1.5 Tri par insertion

Questions :

- Quelle est la complexité du tri par insertion ?

— Dans le `main()`, créez un nouveau tableau `tableauInsertion` en utilisant l'instruction décrite en Section 3.1.3 ;
— Créez une fonction `triInsertion(int[] tableau` et implémentez le tri par insertion ;
— 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.

3.1.6 Tri à bulles

Questions :
— Quelle est la complexité du tri à bulles ?
— Dans le `main()`, créez un nouveau tableau `tableauBulles` en utilisant l'instruction décrite en Section 3.1.3 ;
— Créez une fonction `triBulles(int[] tableau` et implémentez le tri à bulles ;
— 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.

3.1.7 Quick sort

Questions :
— Quelle est la complexité du tri *quick sort* ?
— Dans le `main()`, créez un nouveau tableau `tableauQuickSort` en utilisant l'instruction décrite en Section 3.1.3 ;
— 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.

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

Questions :
— Dans le `main()`, créez un nouveau tableau `tableauSort` en utilisant l'instruction décrite en Section 3.1.3 ;

— Créez une fonction `triJava(int[] tableau)`, et appelez la fonction `Collections.sort()` à l'intérieur de celle-ci ;
— 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.

4 Correction des TP notés

4.1 Remarques générales

— Un TP noté assez difficile qui vous faisait mettre en pratique l'utilisation de structures de contrôles, la manipulation de tableaux, et la modularisation d'un programme.
— Les exercices 1 (encodeur et décodeur de messages) ont dû vous faire travailler sur la compréhension et la conception d'un algorithme générique. Aucun cas particulier n'avait à être traité, sauf celui des caractères `<espace>`.
— Les exercices 2 (morpion et puissance 4) ont dû vous faire travailler sur la compréhension d'un code déjà construit et sur la décomposition d'un programme en plusieurs fonctions.
— Beaucoup d'entre vous ne savent pas lire un énoncé (nom du projet, nom de la classe `Main`).
— Pour info :
  — Résultats des tests de similarité pour les étudiants du lundi : `http://moss.stanford.edu/results/781715821`.
  — Résultats des tests de similarité pour les étudiants du vendredi : `http://moss.stanford.edu/results/386049938/`.
  — Certains TP vont être analysés un peu plus en détails...
  — Les conclusions de ces analyses plus détaillées vous seront transmises directement.

4.2 Encodage et décodage d'un message

Ces deux exercices sont strictement identiques du point de vue algorithmique. Les règles d'encodage et de décodage sont certes différentes, mais la logique reste la même.

Le sujet définit les prototypes des fonctions à implémenter. Ces fonctions (pour l'encodage ou le décodage) prennent un `String` en paramètre et retourne un autre `String`.

Pour cet exercice, on peut d'abord s'intéresser aux valeurs possibles d'entiers que peuvent prendre chaque lettre de nos messages. En regardant l'énoncé, il est écrit que toutes les valeurs doivent être comprises dans l'intervalle [97; 122]. Il est donc nécessaire que notre algorithme produise des valeurs dans cet intervalle, sauf pour les caractères `<espace>` qui sont laissés tels quels.

Ensuite, il faut analyser les règles d'encodage et de décodage afin d'identifier différents cas de figure :

— Les cas simples où l'ajout ou la soustraction de la valeur du décalage suffit (c'est-à-dire, on reste dans l'intervalle [97; 122] ;
— Les cas un peu plus compliqués où l'ajout ou la soustraction de la valeur de décalage ne suffit pas (le décalage nous fait sortir de cet intervalle).

L'idée est donc de trouver la formule permettant de nous assurer que les valeurs en `int` restent dans notre intervalle. Une solution possible est de considérer que chaque lettre est représentée par la formule suivante :

$$lettre = 97 + indicePosition, \quad indicePosition \in [0; 25] \tag{1}$$

, où *indicePosition* correspond à la position de la lettre dans l'alphabet.

Notre problème est réduit à produire une valeur d'*indicePosition* comprise entre 0 et 25. Supposons que nous ayons un entier $ascii \in [97; 122]$ et une valeur de décalage $d > 0$. On a alors :

$$\begin{aligned} & 97 \leq ascii \leq 122 && (2)\\ \Leftrightarrow \quad & 0 \leq ascii - 97 \leq 25 && (3)\\ \Leftrightarrow \quad & d \leq ascii - 97 + d \leq 25 + d && (4) \end{aligned}$$

L'inéquation (4) est problématique puisque si *ascii* est grand (c'est-à-dire, si c'est une lettre proche de la fin de l'alphabet), un décalage positif nous fait sortir de l'intervalle visé [0; 25]. Il faudrait en fait "retourner au début de l'alphabet", et cela est possible en utilisant simplement le `modulo` de 26.

On peut mener un raisonnement similaire avec un décalage d négatif :

$$\begin{aligned} & 97 \leq ascii \leq 122 && (5)\\ \Leftrightarrow \quad & 0 \leq ascii - 97 \leq 25 && (6)\\ \Leftrightarrow \quad & -|d| \leq ascii - 97 - |d| \leq 25 + -|d| && (7) \end{aligned}$$

L'inéquation (7) est désormais problématique puisqu'il est possible d'avoir une valeur négative (si *ascii* est plus petit que la valeur du décalage). Il faudrait en fait aller à la fin de l'alphabet, et cela est possible si on ajoute 26 à tous les membres de l'inéquation, puis que l'on calcule le `modulo` de 26. On note au passage que le fait d'ajouter 26 à l'inéquation (7) ou (4) ne change rien, puisque l'on calcule le `module` de 26 par la suite.

On en déduit donc la formule suivante :

$$indicePosition = (ascii - 97 + 26 + d) \mod 26 \tag{8}$$

Pour conclure, une solution possible à cet exercice est proposée ci-après :

```
public static String codage(String message) {
    String messageCode = "";
    for (int i = 0; i < message.length(); i++) {
        char lettre = message.chartAt(i);
        int ascii = lettre;
        char lettreEncodee;
        if (lettre == ' ') {
            lettreEncodee = ' ';
        } else {
            if (ascii % 2 == 0) {
                // Remplacer d1 et d2 par les valeurs de l'enonce
```

```
                ascii = 97 + ((ascii - 97 + 26 + d1) % 26)
                lettreEncodee = (char) ascii;
            } else {
                ascii = 97 + ((ascii - 97 + 26 - d2) % 26)
                lettreEncodee = (char) ascii;
            }
        }
        messageCode += lettreEncodee;
    }
    return messageCode;
}
```

4.3 Jeux de plateau

Les jeux du Morpion et du Puissance 4 sont très similaires. On notera les différences suivantes :

- La saisie du joueur : deux valeurs pour le Morpion, une seule pour le Puissance 4 ;
- La vérification de la saisie du joueur : on vérifie directement la case du plateau au Morpion, il faut déterminer si la colonne est remplie ou non au Puissance 4 ;
- La mise à jour du plateau : on place le jeton directement dans la case au Morpion, il faut déterminer la bonne "hauteur" au Puissance 4.

Tout le reste n'a pas d'influence sur la conception de notre programme.

4.3.1 Initialisation du plateau

Cette fonction ne devrait présenter aucune difficulté. Il suffit d'imbriquer une boucle `for` dans une autre afin de parcourir le plateau et d'initialiser chaque cellule de ce plateau avec des espaces (`' '`).

4.3.2 Affichage du plateau

Ici, il faut se souvenir que pour afficher le contenu d'un tableau en Java, il est nécessaire de le parcourir et d'en afficher toutes ses valeurs. Dans notre cas, il faut donc encore utiliser des boucles `for` imbriquées pour afficher le plateau.

Les bords du plateau peuvent être affichés en début et fin de chaque ligne. L'utilisation de la fonction `System.out.print()` peut aider à afficher une rangée sans revenir à la ligne.

4.3.3 Saisie du joueur

Ici, il s'agit de répéter la demande de saisie du joueur tant que celle-ci est invalide. Il faut donc s'appuyer sur la fonction `saisieValide()` qui est déclarée juste après.

4.3.4 Saisie valide

Que l'on programme le jeu du Morpion ou du Puissance 4, il faut tout d'abord déterminer si la saisie est un entier compris entre 0 et le nombre de lignes ou de colonnes du plateau.

Ensuite, selon le jeu programmé, il y a une condition supplémentaire à vérifier :

— Pour le Morpion, est-ce que la case définie par la saisie est déjà occupée ?
— Pour le Puissance 4, est-ce que la colonne saisie est déjà remplie ?

Ensuite, il s'agit simplement de renvoyer le résultat de ces tests.

4.3.5 Mise à jour du plateau

Dans le cas du Morpion, il s'agit simplement d'insérer le bon jeton dans la case identifiée par la saisie du joueur. Il faut donc s'appuyer sur la variable `estJoueur1` pour savoir s'il faut ajouter un `O` ou un `X`.

Dans le cas du Puissance 4, il faut dans un premier temps déterminer, pour une colonne donnée, l'indice de ligne le plus grand identifiant une cellule vide. Ensuite, il faut de la même façon utiliser `estJoueur1` pour déterminer s'il faut ajouter un `O` ou un `X`.

4.3.6 Fin du jeu

Le test de fin de jeu est identique quel que soit le jeu programmé :

— Est-ce que le tableau est rempli ? Si oui, le jeu est terminé.
— Est-ce qu'un joueur a créé un alignement gagnant ? Si oui, le jeu est terminé.

Il faut donc s'appuyer sur les fonctions `plateauEstComplet()` et `alignementGagnant()` pour déterminer la fin du jeu.

4.3.7 Plateau complet

Pour déterminer si le plateau est complet, on peut parcourir l'ensemble du plateau et chercher une cellule contenant un espace (`' '`). Si on en trouve un, on peut stopper la boucle en retournant directement la valeur `false`. Si on a parcouru tout le plateau sans jamais trouver d'espace, alors c'est qu'il est rempli.

4.3.8 Alignement gagnant

Bien que les plateaux ne soient pas de même taille et que le nombre de jetons à aligner est différent, l'algorithme qui permet de déterminer si un joueur à gagner ou non est strictement identique pour les deux jeux.

On peut le décomposer en trois étapes :

1. On parcourt toutes les lignes afin de trouver un alignement gagnant en utilisant par exemple une variable `compteur` ;
2. De la même façon, on parcourt toutes les colonnes afin de trouver un alignement gagnant ;
3. On parcourt toutes les diagonales possibles afin de trouver un alignement gagnant.

Le test de détection d'alignement sur les diagonales est légèrement plus complexe à implémenter pour le Puissance 4 puisqu'il y a plus de combinaisons possibles qu'au Morpion. Il faut aussi faire attention aux bornes utilisées dans les boucles permettant de parcourir le plateau.

4.3.9 Logique du jeu

Quel que soit le jeu programmé, la logique du jeu reste la même. Un exemple vous est fourni ci-après afin de vous aider à compléter le reste des fonctions.

```
// La structure est similaire pour le Puissance 4
// Il faut juste faire attention a la difference concernant la saisie
public static void morpion() {
    int choix;
    do {
        System.out.println("1. Jouer\n0. Quitter");
        choix = scanner.nextInt();
        if (choix == 1) {
            boolean jeuFini = false;
            initialiserPlateau();
            while (!jeuFini) {
                afficherPlateau();
                int[] saisie = saisieJoueur();
                updatePlateau(saisie);
                if (finDuJeu()) {
                    jeuFini = true;
                } else {
                    estJoueur1 = !estJoueur1;
                }
            }
        }
    } while (choix == 1);
}
```