Rapport de projet

Projet d'algorithmique

Tris & complexité

Composition :

Nait Djoudi Mounia, Colas Thibaud

Délais :

A rendre pour le lundi 8 mars midi

I. Introduction

Une chaîne industrielle de production d'œufs est constituée de trois machines, qui travaillent le long d'un plateau sur lequel les œufs sont alignés l'un derrière l'autre :

- La première machine dépose en temps réel au fur et à mesure de leur ponte, l'un derrière l'autre, les œufs pondus par une batterie de poules.
- La seconde machine marque les œufs d'une étiquette de couleur et du numéro d'arrivée de l'œuf dans la file. La couleur de l'étiquette est bleu pour les gros œufs, blanc pour les moyens, et rouge pour les petits.
- Une fois le plateau rempli, la troisième machine a pour fonction de trier les œufs: elle place les gros (bleus) en tête, les moyens (blancs) ensuite, et les petits (rouges) en queue. Les œufs non encore triés se situent entre les moyens et les petits.

Ici, nous allons nous occuper de modéliser le tri dont il est question, d'évaluer sa complexité et d'en développer des versions plus efficaces.

II. Questions

a. Algorithme en pseudo-langage

```
Fonction trilteratif retourne Entier
        Paramètres : Ponte(D/R) : File
        Variables : comp : entier
                    tmp : Oeuf
Début
        comp -> 0
        Tant que (Ponte.finBlanc < Ponte.finPioche) Faire
                Si (Ponte.tableau[finBlanc+1].couleur = « bleu ») Alors
                        tmp <- Ponte.tabOeuf[finBlanc+1]
                        Pour i <- finBlanc à finBleu Faire
                                tabOeuf[i+1] <- tabOeuf[i]
                                comp <- comp + 1
                                i <- i -1
                        fpour
                        Ponte.tabOeuf[nbOeufs-1] <- tmp
                        Comp <- comp+1
                        finBleu <- finBleu + 1
                        finBlanc <- finBlanc + 1
                Sinon Si (Ponte.tableau[finBlanc+1].couleur= « blanc ») Alors
                        finBlanc <- finBlanc + 1
                Sinon Si (Ponte.tableau[finBlanc+1].couleur= « rouge ») Alors
                        tmp <- Ponte.tabOeuf[finBlanc+1]
                        Pour i <- finBlanc + 2 à nbOeufs Faire
                                tabOeuf[i-1] <- tabOeuf[i]
```

```
                    comp <- comp + 1
                    i <- i + 1
                fpour
                Ponte.tabOeuf[nbOeufs-1] <- tmp
                Comp <- comp+1
                finPioche <- finPioche – 1
            Fsi
        Ftq
        Retourne comp
Fin
```

b. Implantation en Java

L'implantation est effectuée et tout fonctionne. Elle est divisée en deux classes : Œuf et File. Pour évaluer la complexité de l'algorithme de tri, on calcule le nombre d'écritures dans le tableau et on le renvoie. La durée d'exécution est calculée par la fonction duréeIteratif().

c. Sous-programme de test

La solution à cette question prend la forme d'une fonction renvoyant un booléen : verifTri() renvoie vrai si le tableau est trié, et faux sinon. On vérifie l'ordre croissant des numéros des œufs ainsi que l'ordre de leurs couleurs.

d. Génération aléatoire

On gère le choix aléatoire de la taille de l'œuf (de sa couleur) grâce à `Math.random()*3`

La taille maximale du tableau reste inconnue mais on sait que le programme continue de fonctionner avec une taille de 300'000 œufs.

e. Programme principal

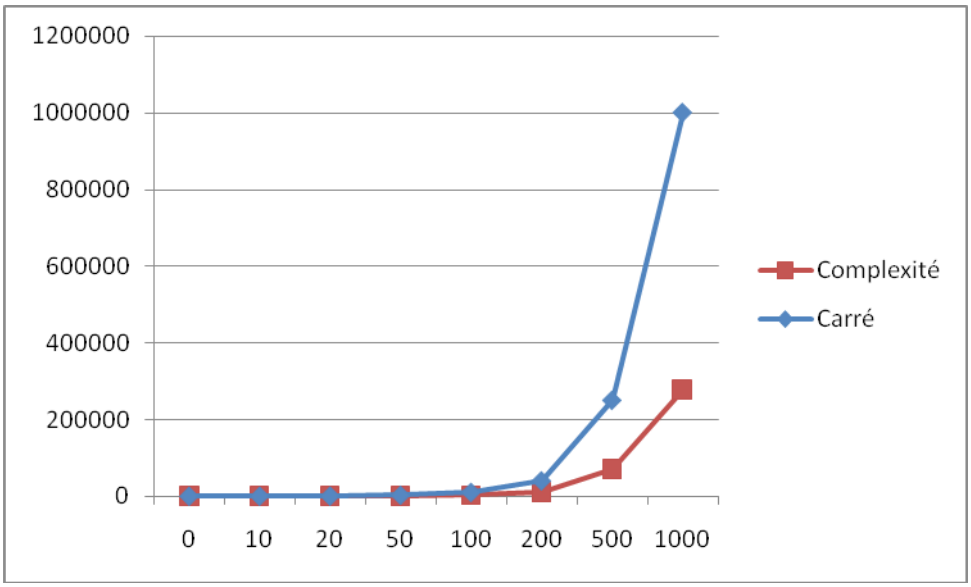
Le programme principal demande à l'utilisateur de saisir une taille de tableau pour la File qui va être générée et triée 1000 fois de suite, puis il effectue et vérifie le tri 1000 fois. Exemple :

```
Quelle taille voulez-vous pour votre tableau ?
200
1 trié.
2 trié.
3 trié.
4 trié.
5 trié.
6 trié.
7 trié.
8 trié.
9 trié.
10 trié.
```

f. Calcul de la complexité moyenne

On calcule la complexité moyenne de l'algorithme en l'appelant 1000 fois sur 1000 tableaux non triés et en faisant la moyenne des nombres d'écriture.

Taille	Complexité	Carré
0	0	0
10	32,097	100
20	120,612	400
50	712,384	2500
100	2802,65	10000
200	11203,726	40000
500	69719,061	250000



Complexité = 0.28 * n^2

g. Implantation et test d'une version récursive

Il suffit de légèrement modifier la fonction de tri pour l'exécuter récursivement. Au dessus d'une taille de 4721, la procédure de tri ne fonctionne plus.

h. Comparaison avec la version itérative

La version récursive reprend quasiment le même code à deux lignes près et ne permet aucun gain en temps d'exécution ni en utilisation du processeur, elle est même plus lente et demande plus de mémoire libre. La pile est saturée avec un tableau de 4721 valeurs alors que la version itérative permet d'atteindre des centaines de millions de valeurs.

i. Méthode plus efficace

Puisqu'on profite d'un second tableau pouvant contenir les œufs triés, on va l'utiliser pour notre algorithme de tri. On va parcourir la File une fois pour repérer tous les bleus, les copier dans le deuxième tableau, la parcourir une deuxième fois pour repérer et copier les blancs puis une troisième et dernière fois pour tous les rouges. Soit trois parcours de tableau et un nombre fixe d'écritures = n.

j. Implantation de cette méthode

Cette nouvelle méthode révolutionnaire a une complexité égale à n. Dans le meilleur des cas (tableau déjà trié), cette méthode est moins efficace que le tri précédent, mais elle est bien plus efficace dans le cas général et dans le pire des cas. Implantée dans la méthode triSuperCool(File pleinDoeufs).

III. Conclusion

Ce projet nous a permis de mettre en œuvre nos connaissances en matière de tris et de conversion d'une procédure d'une version itérative à une version récursive. Nous nous sommes rendus compte de la supériorité de la méthode itérative sur la méthode récursive.