

Mini-projet 2 - Conception d'algorithme : Troycount

1. Explication de l'algorithme

1.1 Procédure

1. Créer un tableau de personne (Voir figure 2)
2. Ordonner ce tableau en fonction de la balance financière de chaque personne
3. Séparer le tableau en 2
 - Personnes ayant une balance positive (= débiteurs)
 - Personnes ayant une balance négative (= créanciers)

NB : Les personnes ayant une balance nulle sont enlevées (Elle ne doivent pas donner/recevoir d'argent)
4. Parcourir le tableau de débiteurs et de créanciers
5. A chaque itération, effectuer les transactions appropriées (Voir 1.2 gestion des transactions)

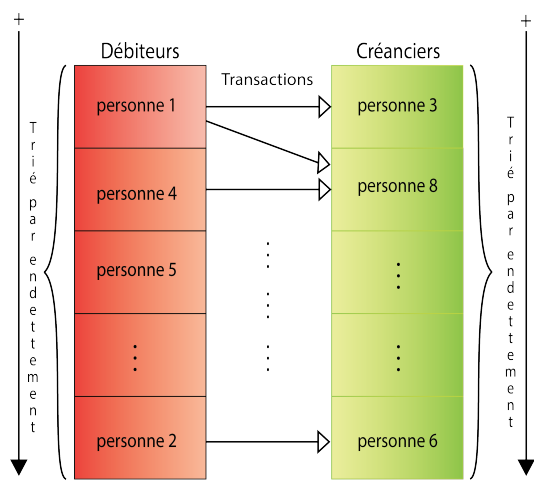


FIGURE 1 – Création des transactions

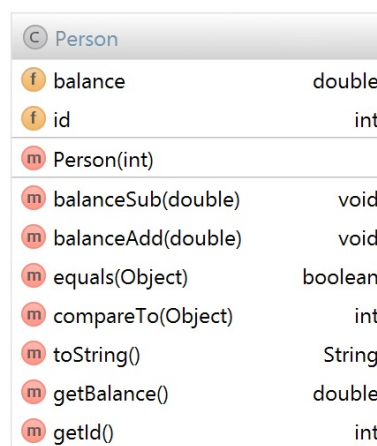


FIGURE 2 – Objet "personne"

1.2 Gestion des transactions

Afin de gérer les transactions, les 2 tableaux sont parcourus simultanément (personnes créancières et personnes débitrices)

Fonctionnement

1. Commencer au début de chaque tableau
2. Comparer le débiteur actuel avec le créancier actuel ; 2 situations :
 - (a) le débiteur actuel peut rembourser intégralement le créancier actuel
 - i. Le débiteur lui verse toute la créance
 - ii. Passer au créancier suivant
 - (b) le débiteur ne peut rembourser intégralement le créancier actuel
 - i. Le débiteur lui verse toute sa dette
 - ii. Passer au débiteur suivant
3. Recommencer l'étape 2 jusqu'à la fin des tableaux

NB : Les transactions sont stockées au fur et à mesure dans une pile. Celle-ci est ensuite retournée sous forme d'un tableau

2. Complexité des différentes méthodes

Soit n , le nombre de personnes : n représente donc la taille du problème à résoudre.

2.1 `Troycount.balance()`

Complexité temporelle en $\Theta(n)$ car :

1. L'expression sera toujours parcourue entièrement 1 fois, à l'aide d'une boucle.
2. La boucle principale ne contient que des opérations de complexité $\Theta(1)$.

On en déduit donc que le temps d'exécution augmentera de façon linéaire par rapport à la taille du problème.

2.2 `Troycount.generateTabPerson`

Complexité temporelle en $\mathcal{O}(\log(n))$ car :

1. Complexité temporelle en $\mathcal{O}(h)$, où h est la hauteur de l'arbre
2. L'arbre est équilibré donc $h \in \Theta(\log_2(n))$

Donc,

- Meilleur cas : $\Theta(1)$ (on tombe sur la racine)
- Pire cas : $\Theta(\log(n))$
- En général : $\mathcal{O}(\log(n))$

3. Choix d'implémentation

Afin d'implémenter notre algorithme nous avons utilisé la programmation orienté objet ; Une classe "personne" a été créée afin de gérer plus facilement les créances et dettes de chaque personne. De plus, cette classe "personne" implémente l'interface "Comparable" ce qui permet de trier facilement les créanciers et les débiteurs.

Annexe

Troycount.balance()

```
1 public Transaction[] balance() {
2     Person[] persons = generateTabPersons();
3     Arrays.sort(persons);
4
5     int firstSplitIndex = 0;
6     int secondSplitIndex = 0;
7
8     for (int i = 0; persons[i].getBalance() >= 0; i++) {
9         secondSplitIndex++;
10        if (persons[i].getBalance() > 0) {
11            firstSplitIndex++;
12        }
13    }
14
15    Person[] debtors = Arrays.copyOfRange(persons, 0, firstSplitIndex);
16    Person[] creditors = Arrays.copyOfRange(persons, secondSplitIndex,
17        persons.length);
18
19    reverse(creditors);
20
21    Stack<Transaction> transactions = new Stack<Transaction>();
22    int i = 0, j = 0;
23
24    while (i < debtors.length && j < creditors.length) {
25        if (debtors[i].getBalance() <= -creditors[j].getBalance()){
26            double transactionAmount = debtors[i].getBalance();
27            Transaction t = new Transaction(debtors[i].getId(),
28                creditors[j].getId(),
29                transactionAmount);
30
31            transactions.add(t);
32            System.out.println(t);
33            creditors[j].balanceAdd(transactionAmount);
34            debtors[i].balanceSub(transactionAmount);
35            i++;
36        } else {
37            double transactionAmount = -creditors[j].getBalance();
38            Transaction t = new Transaction(debtors[i].getId(),
39                creditors[j].getId(),
40                transactionAmount);
41
42            transactions.add(t);
43            System.out.println(t);
44            creditors[j].balanceAdd(transactionAmount);
45            debtors[i].balanceSub(transactionAmount);
46            j++;
47        }
48    }
49    return transactions.toArray(new Transaction[transactions.size()]);
50 }
```

Troycount.generateTabPerson()

```
1 public Person[] generateTabPersons() {
2     Person[] persons = new Person[group_size];
3     for (int i = 0; i < group_size; i++) {
4         int personID = i + 1;
5         persons[i] = new Person(personID);
6     }
7     for (Spending aSpending : spendings) {
8         int debitedPerson = aSpending.get_paid_by();
9         int[] chargedPerson = aSpending.get_paid_for();
10        int chargedPersons = chargedPerson.length;
11        double spendingAmount = aSpending.get_amount();
12
13        for (int person : chargedPerson) {
14            double fixedCharge = aSpending.get_fixed_charges(person);
15            spendingAmount -= fixedCharge;
16        }
17
18        for (int person : chargedPerson) {
19            persons[person - 1].balanceAdd(spendingAmount / chargedPersons);
20        }
21
22        persons[debitedPerson - 1].balanceSub(spendingAmount);
23    }
24    return persons;
25 }
```