Sacilotto Nicolas (3B)

Programmation efficace

TP#4 Un TP au pénitencier

Table des matières

Programmation efficace

TP#4 Un TP au pénitencier

1 Casser une pierre en deux

L'opération cassant une pierre en deux est prénommée `split()`. La signature algorithmique de cette opération est la suivante :

`Stone` → `Stone`

La méthode Java possède effectivement un effet de bord puisque les caractéristiques de la pierre initiale changent (son poids et également son diamètre) sans pour autant créer deux objets de type 'Stone' qui eux, auraient pour caractéristiques, les caractéristiques de la pierre initiale coupée.

```java
import stone.Stone;
public class exo1 {
    public static void main(String[] args) {
     Stone laPierre = Stone.makeSmallStone();
     System.out.println(laPierre.toString());
     Stone leMorceau1 = laPierre.split();
     System.out.println("Première pierre : " + leMorceau1.toString() + "\nDeuxième pierre :" + laPierre.toString());
    }
}
```

2 User une pierre jusqu'à un certain « diamètre »

Pour le nombre d'opérations `split()` a effectuer en fonction du poids de la pierre, la classe de complexité obtenue est une classe de O(log(n)) car a chaque appel de la méthode `split()`, le poids de la pierre est approximativement divise par deux. De plus, augmenter le poids de la pierre par 2 ne prendra seulement qu'une opération supplémentaire.

```
    Stone laPierre = Stone.makeHugeStone();
    Stone leMorceau;
    leMorceau = laPierre.split();
    int i = 0;
    int compteur = 0;
    do {
        System.out.println(leMorceau.diameter());
        compteur ++;
        leMorceau = laPierre.split();
    } while (leMorceau.diameter() > 5);
    System.out.println("Nombre d'opérations : " + compteur);
}
```

3 Fragmenter une pierre jusqu'à un certain « diamètre »

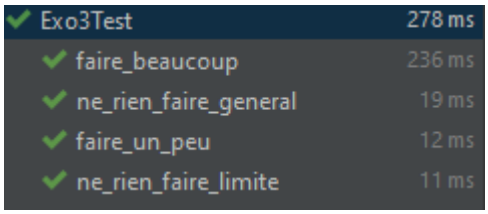
La classe de complexité pour le nombre d'éléments de la collection en fonction du poids de la pierre est d'une complexité de O(n). Concrètement, une grosse pierre est 200x plus lourde qu'une petite pierre (5kg*200 = 1000kg). Lorsque l'on vient a multiplier la taille de la liste (correspondant a l'objet retourne de la fonction `grind()`) pour une petite pierre, par 200, celle-ci correspond approximativement a la taille réelle de la liste obtenue pour une grosse pierre (le test étant réalise avec un diamètre de 10cm). Évidemment, tout cela reste approximatif car `split()` ne divise pas exactement, équitablement, le poids des deux pierres.

```
@Override
public Collection<Stone> grind(Stone stone, int diameter) {
    List<Stone> alPierre = new ArrayList<Stone>();
    alPierre.add(stone);

    for(int i = 0; i < alPierre.size(); i++) {
        while(alPierre.get(i).diameter() > diameter) {
            alPierre.add(alPierre.get(i).split());
        }
    }
    return alPierre;
}
```

Puis le test :

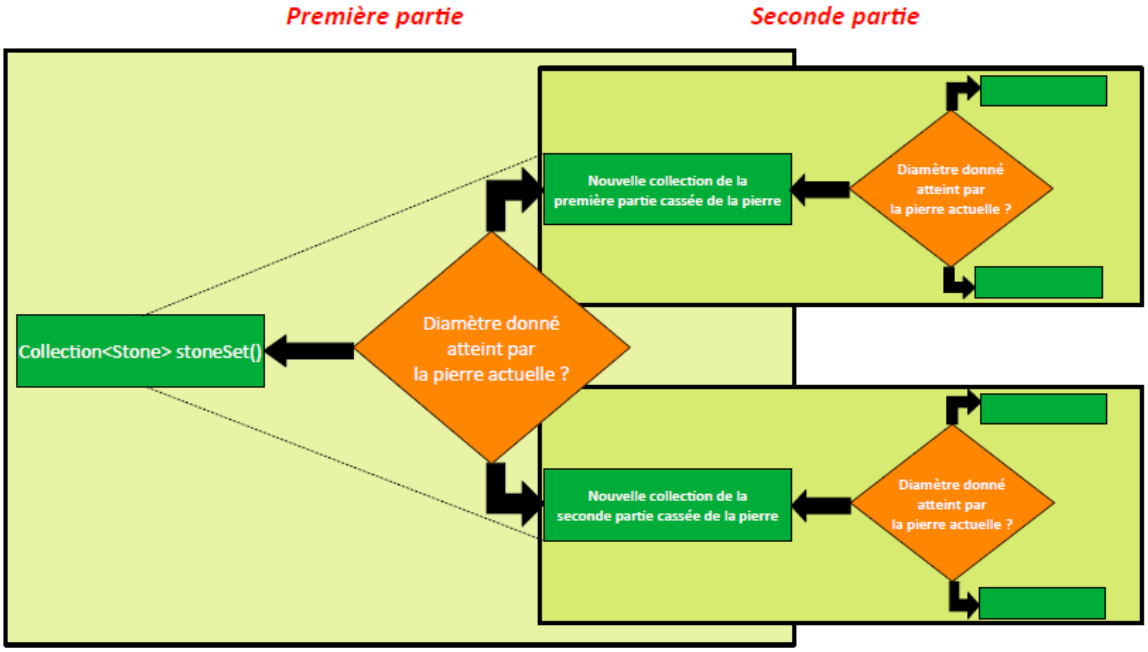
```java
import stone.AbstractGrinderTest;
import stone.Grinder;

public class Exo3Test extends AbstractGrinderTest {

    @Override
    protected Grinder makeGrinder() { return new MyGrinder(); }

}
```

les test :



4 description des structures de données employées

En voici le schéma illustratif au cours de l'exécution de ma méthode `grind()` :



Chaque partie se répète jusqu'à ce que le diamètre de la pierre actuelle corresponde au diamètre donne.

Procédé récursif de l'état de la collection (Le tableau doit se lire à partir du bas)
[MainStone]
[MainStone, SplittedStone, SplittedStone]
[MainStone, SplittedStone, SplittedStone, FirstPartSplittedStone, FirstPartSplittedStone, SecondPartSplittedStone, SecondPartSplittedStone]
[…]
[…]
[MainStone, …, InitialStone]

Concrètement, l'algorithme se déroule de la manière suivante :

- A partir d'une pierre initiale passée en paramètre (peu importe de sa taille, même si elle aura son influence par rapport au diamètre fourni), on vérifie si le diamètre donne est supérieur au diamètre de la pierre passée dans la collection.
- Si c'est le cas, on ajout directement la pierre dans la collection puisqu'elle peut rentrer dans le trou.
- Si ce n'est pas le cas, le processus est de couper en deux morceaux, la pierre, dans deux collections différentes qui viendront être ajoutées a la collection mère.

Ainsi, ce processus est récursif puisqu'à chaque fois que le diamètre de la pierre n'est pas conforme, on répète ce même processus. La méthode `addAll()` permet ici, de concaténer des tableaux qui seront réutilises dans leur propre processus.

5 Passage de récursif à non récursif

```
3 usages
public Collection<Stone> notRecursiveGrind(Stone stone, int diameter) {
    Collection<Stone> listOfStone = new ArrayList<Stone>();
    Collection<Stone> tempListOfStone;

    Stone stoneTemp;
    boolean splitted = true;

    listOfStone.add(stone);

    while(splitted) {
        splitted = false;
        tempListOfStone = new ArrayList<Stone>();

        for(Stone stones : listOfStone) {
            while(stones.diameter() > diameter) {
                splitted = true;
                stoneTemp = stones.split();
                tempListOfStone.add(stoneTemp);
            }
        }
        listOfStone.addAll(tempListOfStone);
    }

    return listOfStone;
}
```

En voici le nombre d'appels pour la fonction `grind()` récursive :