Alexis BERTRAND
Guillaume BRUCHON

TP CS353 - Algorithmique

TP5 : Table de Hachage

Pour une question de lisibilité du débug, la taille de la table de hash a été réduit à 13 éléments, mais 1009 fonctionne parfaitement (le main de test s'adapte)

Exercice 1 : Fonction de hachage

Simple traduction d'équations mathématiques :

```
int hashkey(int itemCode,int nbTry)
{
    int h1 = itemCode % TABLE_SIZE;
    int h2 = 1 + (itemCode % (TABLE_SIZE-1));
    return (h1 + nbTry * h2) % TABLE_SIZE;
}
```

Exercice 2 : Fonction d'insertion

L' insertion à été réalisé de manière récursive en 3 fonctions : une fonction utilisateur, et deux fonction de récursion.

Fonction utilisateur (non récursive)

```
int insertItem(int itemCode, char* itemName, float itemPrice) {
    Item k;
    k.code = itemCode;
    int i = strlen(itemName); // copie du nom de l'objet et ajout
d'espaces (pour l'affichage)
    memcpy(k.name,itemName,((i>32)?32:i)*sizeof(char));
```

```
        for (; i<31; i++)
            k.name[i] = ' ';
        k.name[i] = '\0';
        k.price = itemPrice;
        k.dirty = false;
        return insRec(k,0); //appel de la fonction d'insertion récursive
    }
```

On applique au nom un traitement pour le compléter avec des espaces dans le but de garder l'affichage aligné.

Fonction d'insertion récursive :

```
    int insRec(Item k, int try) {
        if(try >= TABLE_SIZE) return TABLE_FULL; //cas du tableau pleins
        int i = hashkey(k.code,try);
        if(hash_table[i].code == k.code) return INSERT_ALREADY_EXIST; //cas
    élément déja existant
        if(hash_table[i].code == NULL_ITEM) { //cas insertion direct
            hash_table[i] = k;
            return SUCCESS;
        }
        else if(hash_table[i].code == DELETED_ITEM) {//cas case supprimée,
    un des 2 cas précédants
            if(present(k,try) == 1) return INSERT_ALREADY_EXIST; //recherche
    de la présence de l'élément et cas existant
            else {
                hash_table[i] = k; //sinon insertion sur ce DELETED_ITEM
                return SUCCESS;
            }
        }
        else {
            return insRec(k,try+1); //sinon on sonde un autre emplacement
        }
    }
```

On remarque que l'appel récursif se fait toujours en dernière instruction, ce qui rend possible l'optimisation **tail-recursion** par le compilateur (mais pas forcément réalisé par tous les compilateurs). Dans ce cas, le compilateur transforme l'appel récursif en une simple boucle et on ne consomme plus de stack pour rien (on évite aussi les stack overflow).

Fonction `présence()` :

```
    int present(Item k, int try) {
      if(try >= TABLE_SIZE) return 0;
        int i = hashkey(k.code,try);
        if(hash_table[i].code == k.code) return 1;
        if(hash_table[i].code == NULL_ITEM) return 0;
```

```
        else return present(k,try+1);
    }
```

Cette fonction permet de remonter le sondage d'une clée pour verifier que l'element ne s'y trouve pas déja en cas d'élément supprimé.
Elle est aussi **tail récursive**.

Exercice 3 : Suppression

La fonction de suppression utilise une fonction de recherche d'item qui est utilisé dans d'autres fonction et qui retourne l'indice d'un éliment dans la table :

```
int suppressItem(int itemCode)
{
    int indice = searchItem(itemCode);
    if (indice != -1) {
        hash_table[indice].code = DELETED_ITEM;
            hash_table[indice].price = 0.00f;
        return SUCCESS;
    }

    return DELETE_NO_ROW;
}
```

La fonction de recherche réalise un sondage jusque à trouver l'élément ou NULL_ITEM. Elle parcours la table de hash jusqu'à trouver l'élément et retourne -1 si l'élément n'existe pas.

```
int searchItem(int itemCode) {
    int i = 0, hash;
    do {
        hash = hashkey (itemCode, i++);
        if (hash_table[hash].code == itemCode)
            return hash;
    } while (hash_table[hash].code != NULL_ITEM && i<TABLE_SIZE);
    return -1;
}
```

Exercice 4 : Affichage

Nous effectuons un affichage par ordre d'index dans le tableau (nous avions une version qui affichait dans l'ordre des clés mais ce n'était pas optimisé et assez long).

```
void dumpItems()
{
    int i;
    printf("CODE\tLIBELLE\t\t\tPRIX\tINDEX\n");
    for (i=0; i<TABLE_SIZE; i++) {
        if (hash_table[i].code >= 0)
            printf("%d\t%s\t%0.2f\t%d\n", hash_table[i].code,
hash_table[i].name, hash_table[i].price, i);
    }
}
```

Exercice 5 : Prix

On utilise encore `searchItem()` , rien de particulier

```
float getPrice(int itemCode)
{
    int indice = searchItem(itemCode);
    if (indice != -1)
        return hash_table[indice].price;
    return SELECT_NO_ROW;
}
```

Exercice 6 : Update

On utilise encore `searchItem()` pour trouver l'élément, puis on met à jour son prix et son nom. On applique au nouveau nom le même traitement que pour l'insertion

```
int updateItem(int itemCode, char* itemName, float itemPrice)
{
    int indice = searchItem(itemCode);
    int i = strlen(itemName);

    if (indice != -1) {
        memcpy(hash_table[indice].name,itemName,((i>32)?
32:i)*sizeof(char));
        for (i=strlen(itemName); i<31; i++)
            hash_table[indice].name[i] = ' ';
        hash_table[indice].name[i] = '\0';
        hash_table[indice].price = itemPrice;
        return SUCCESS;
    }
```

```
    return UPDATE_NO_ROW;
}
```

Exercice 7 : Rebuild Table

Encore une fois, la fonction de rebuild inSitu a été réalisé de manière récursive.

Fonction utilisateur:

```
void rebuildTable()
{
    int i;
//Au début, tout les éléments sont sales
    for (i=0; i<TABLE_SIZE; i++)
        if (hash_table[i].code !=  NULL_ITEM)
            hash_table[i].dirty = true;

//On nettoie tout les éléments
    for (i=0; i<TABLE_SIZE; i++) {
    if(hash_table[i].dirty == true) {
        if (hash_table[i].code >= 0 ) {
        Item k = hash_table[i];
        hash_table[i].code = NULL_ITEM;
            insertRebuild(k,0);
      }
      else
        hash_table[i].code = NULL_ITEM;
    }
    }
}
```

On commence par taguer tous les éléments en **dirty**
Ensuite on “nettoie” tous les éléments non vides et dirty grace à une procédure récursive qui suit (on en profite aussi pour transformer les DELETED_ITEM en NULL_ITEM)

Fonction de réinsertion Propre :

```
void insertRebuild(Item k, int try) {
  int hash = hashkey(k.code,try);
  //cas place libre
  if(hash_table[hash].code == NULL_ITEM || hash_table[hash].code ==
DELETED_ITEM) {
    hash_table[hash] = k;
    hash_table[hash].dirty = false;
  }
```

```
  //cas emplacement déja occupé mais sale
  else if(hash_table[hash].dirty == true) {
    Item newK = hash_table[hash];
    hash_table[hash] = k;
    hash_table[hash].dirty = false;
    insertRebuild(newK,0);//on replace le nouvel élément
  }
  //cas emplacement occupé et propre
  else {
    insertRebuild(k,try+1);//on sonde un autre emplacement
  }
}
```

Cette fonction replace l'élément avec un sondage le plus faible possible et le tag en "propre". Si elle tombe sur un emplacement sale, elle y place l'élément courant et replace l'élément anciennement présent de manière récursive (tail récursion).

Exercice 8 : Recherche par libellé

La recherche par libellé se fait avec la fonction `Item* findItem(char* itemName)` On remarque que le prototype proposé manque l'information sur la taille du tableau, elle est donc pas utilisable en pratique.

```
Item* findItem(char* itemName)
{
  char formatedName[32]; //comme pour l'insertion et la mise à jour, on
met en forme la string pour la comparaison
  int i = strlen(itemName);
    memcpy(formatedName,itemName,((i>32)?32:i)*sizeof(char));
    for (; i<31; i++)
        formatedName[i] = ' ';
    formatedName[i] = '\0';

  Item* found = malloc(0);//on initialise le pointeur de retour pour
utiliser realloc
  if (found == NULL) {
    fprintf(stderr, "erreur malloc\n");
    exit(1);
  }

  int j;
  int k;
  for (j=0,k=0; j<TABLE_SIZE; j++) {
    //on filtre les éléments à garder
        if (hash_table[j].code >= 0 &&
strcmp(hash_table[j].name,formatedName) == 0) {
          found = realloc(found,(++k)*sizeof(Item)); //réservation d'une
```

```
place pour l'élément trouvé
      found[k-1] = hash_table[j];//ajout de l'element à la fin
    }
  }
  if(k == 0) {
    free(found);
    return NULL;
  }
  else {
    return found;
  }
}
```

Pour rendre cette fonction utilisable en pratique il aurait fallu par exemple passer un pointeur sur entier à la fonction pour pouvoir recevoir le nombre d'objets trouvés (ou renvoyer cet entier et passer un double pointeur en paramètre pour le tableau)

Exercice 9 : table des libellés

Cette partie n'a pas été traité, mais pour la réaliser il aurait fallu créé une autre table de hash de même taille en globale et implémenter les fonctions d'insertion et de suppression correspondante (similaires), puis les utiliser dans les fonctions réalisés plus haut quand nécessaire. `Item* findItem(char* itemName)` aurait ainsi pu être optimisé.

Exécution

```
****** Gestionnaire de magasin ******
CODE    LIBELLE                         PRIX    INDEX
12883   itemRand code: 12883            128.83  0
14268   itemRand code: 14268            142.68  1
16287   itemRand code: 16287            162.87  2
16483   itemRand code: 16483            164.83  3
2786    itemRand code: 2786             27.86   4
6317    itemRand code: 6317             63.17   5
760     itemRand code: 760              7.60    6
13394   itemRand code: 13394            133.94  7
26463   itemRand code: 26463            264.63  8
16233   itemRand code: 16233            162.33  9
3910    itemRand code: 3910             39.10   10
13440   itemRand code: 13440            134.40  11
30413   itemRand code: 30413            304.13  12
=======Gestionnaire de magasin ======
=======Suppression aléatoire de la moitié des éléments ======
CODE    LIBELLE                         PRIX    INDEX
```

```
760     itemRand code: 760              7.60    6
13394   itemRand code: 13394            133.94  7
26463   itemRand code: 26463            264.63  8
16233   itemRand code: 16233            162.33  9
3910    itemRand code: 3910             39.10   10
13440   itemRand code: 13440            134.40  11
30413   itemRand code: 30413            304.13  12
=======Réinsertion des éléments ======
CODE    LIBELLE                         PRIX    INDEX
20214   itemRand code: 20214            202.14  0
4443    itemRand code: 4443             44.43   1
4735    itemRand code: 4735             47.35   2
1329    itemRand code: 1329             13.29   3
763     itemRand code: 763              7.63    4
7959    itemRand code: 7959             79.59   5
760     itemRand code: 760              7.60    6
13394   itemRand code: 13394            133.94  7
26463   itemRand code: 26463            264.63  8
16233   itemRand code: 16233            162.33  9
3910    itemRand code: 3910             39.10   10
13440   itemRand code: 13440            134.40  11
30413   itemRand code: 30413            304.13  12
=======Suppression de tous les éléments ======
=======Réinsertion de la moitié des éléments avec seed 42 ======
CODE    LIBELLE                         PRIX    INDEX
16287   itemRand code: 16287            162.87  2
4242    item à mettre à jour            9999.00 4
760     itemRand code: 760              7.60    6
26463   itemRand code: 26463            264.63  8
16233   itemRand code: 16233            162.33  9
13440   itemRand code: 13440            134.40  11
30413   itemRand code: 30413            304.13  12
*******Utilisation de getPrice pour l'item 4242 ******
Le Prix de l'item 4242 est 9999.00
Mise à jour de cet item via updateItem(item, "item mis à jour",
42.4242), Code de retour:0
Le Prix de l'item 4242 est 42.42, son nom est "item à mettre à jour
"
CODE    LIBELLE                         PRIX    INDEX
16287   itemRand code: 16287            162.87  2
4242    item mis à jour                 42.42   4
760     itemRand code: 760              7.60    6
26463   itemRand code: 26463            264.63  8
16233   itemRand code: 16233            162.33  9
13440   itemRand code: 13440            134.40  11
30413   itemRand code: 30413            304.13  12
*******Affichage des métadonnées de la table*****
CODE    PRIX    INDEX   DIRTY
-2      0.00    0       false
-2      0.00    1       false
16287   162.87  2       false
```

```
-2      0.00    3       false
4242    42.42   4       false
-2      0.00    5       false
760     7.60    6       false
-2      0.00    7       false
26463   264.63  8       false
16233   162.33  9       false
-2      0.00    10      false
13440   134.40  11      false
30413   304.13  12      false
*******Reconstruction de la table inSitu*********
CODE    PRIX    INDEX   DIRTY
-1      0.00    0       true
-1      0.00    1       true
-1      162.87  2       true
760     7.60    3       false
4242    42.42   4       false
-1      0.00    5       true
30413   304.13  6       false
-1      0.00    7       true
26463   264.63  8       false
16233   162.33  9       false
-1      0.00    10      true
16287   162.87  11      false
13440   134.40  12      false
CODE    LIBELLE                         PRIX    INDEX
760     itemRand code: 760              7.60    3
4242    item mis ├á jour                42.42   4
30413   itemRand code: 30413            304.13  6
26463   itemRand code: 26463            264.63  8
16233   itemRand code: 16233            162.33  9
16287   itemRand code: 16287            162.87  11
13440   itemRand code: 13440            134.40  12
*******Test de findItem(), ajout d'un deuxième "item mis é jour" et
recherche de ces deux items
CODE    LIBELLE                         PRIX
314     item mis à jour                3.14
4242    item mis à jour                42.42
```

TP6 : Rainbow Table

Les Rainbow Table (ou tables arc en ciel) sont des outils permettant de casser des mots de passe. En effet, il n'existe pas d'opération afin de décrypter des mots de passe (cryptés en md5 par exemple). Ainsi, si l'on souhaite décoder ces mots de passe, il faut tester toutes les combinaisons possible (Brute force). Afin de ne pas avoir à le refaire à chaque fois, on peut stocker toutes les clés crypté avec le mot de passe correspond dans une table. Cependant cela peut prendre beaucoup de place. Les Rainbow table permettent de compresser ces tables de décodage pour les rendre plus "utilisable".

Exercice 1 : La force brute

On utilise le programme `hashcat` avec les option :

- a 3 : pour le mode bruteforce
- m 0 : pour codage MD5
- la règle ?d?d?d?d?d?d signifiant 6 caractères décimaux

```
root@kali:~# hashcat -m 0 -a 3 Downloads/hash.txt ?d?d?d?d?d?d
Initializing hashcat v2.00 with 4 threads and 32mb segment-size...

Added hashes from file Downloads/hash.txt: 4 (1 salts)

84e6a804e2069365df19ab2d0157e818:712333
7c436f13e2c4e8309a93d1e1c887a228:799001
c63b2b0396dd870448894f1152320e67:845333
c3a222a452fe95c956953c41f46cd334:712456

All hashes have been recovered

Started: Wed Apr 27 15:04:41 2016
Stopped: Wed Apr 27 15:04:42 2016
```

Comme on peut le voir, le temps de calcul est assez rapide : 1 sec. Cela vient du faite que les mots de passe ont un formatage bien défini : 6 chiffres, ce qui ne laisse que 1 000 000 possibilités.

Résultats :

- Alice : 712333
- Clara : 799001
- Dilbert : 845333
- Bob : 712456

Exercice 2 : Calcul d'une chaine

On défini la fonction `CalculChaine` qui permet de calculer à partir d'un mot de passe, plusieurs hash de mot de passe différents. **En stockant uniquement le début et la fin de cette chaine, on sera alors en mesure de retrouver tout les mots de passe qui ont étés calculés dans la chaine.** Ce qui permet de gagner de la place.
Comme on peut le voir sur l'exemple ci-dessous, on calcule le Hash du mot de passe, puis on le réduit afin d'obtenir un nouveau mot de passe, que l'on va à nouveau hasher, …

La fonction retournera le dernier élément de la chaine après 999 hash et réduction.

On défini alors des fonctions auxiliaires qui permettent de réaliser ces actions :

- La fonction `reduction` qui réduit une chaine md5 en un mot de passe de 6 caractères :

```
public int reduction(byte[] hash, int num) {
    int res = num;
    int mult = 1;
    for (int i = 0; i < 4; i++) {
        res = res + mult * hash[i];
        mult = mult * 256;
    }
    if (res < 0) {
        res = -res;
    }
    res = res % 1000000;
    return res;
}
```

- Une fonction `md5` qui permet de transformer un mot de passe de 6 caractères en String puis de le crypter en md5 :

```
public byte[] md5(int px) {
    private MessageDigestdigest = MessageDigest.getInstance("MD5");
    return digest.digest(String.format("%06d", px) .getBytes());
}
```

Il ne reste alors plus qu'à assembler ces deux parties dans la fonction qui, à partir de P0, calcule P999. Elle calcule successivement le md5 du mot de passe, puis la réduction de ce hash afin d'avoir un nouveau mot de passe à crypter, …

```
public int CalculChaine (int i, int px) {
    byte[] hash;
```

```
    for ( ; i<1000; i++) {
        hash = md5(px);
        px = reduction(hash, i);
    }
    return px;
}
```

> Remarque : La fonction prend en faite en entrée Pi et l'indice i. Cet indice i correspond à l'indice de la première réduction à appliquer. En fait pour l'instant on aura toujours i=0 car on part du début de la chaine. Ce paramètre nous servira pour le décodage de la table, quand on calculera le hash pour différentes réductions.

Si on teste nos méthode avec le mot de passe 1, on trouve bien :

```
ChaineCalcul(1) = 279947
```

Exercice 3 : Calcul de la table complète

Question préliminaire

1. Pourquoi calcule-t-on 10 000 chaines?
   Il y a 1 000 000 possibilités pour le mot de passe, et on réalise des chaines de 100 mots de passes, il faut donc en théorie une table avec 10 000 chaines distinctes. Or la table est remplie de manière aléatoire et il peut y avoir des croisements de chaines (mais pas de doublons) il faut donc augmenter le nombre de chaines pour augmenter la probabilité de réussite. 10 000 est un bon compromis entre temps de génération et probabilité de réussite.

2. Pourquoi utilise t on la valeur 10 061 pour la table de hachage ?
   10 061 est le premier nombre premier après 10 000, et pour que la fonction de hachage de la table agisse bien comme une permutation, il est nécessaire d'utiliser un nombre premier.

3. Combien y a t il réellement d'éléments dans la table ? Pourquoi ?
   On verra par la suite qu'il n'y a que 1612 éléments dans la table (cela peut varier d'une génération à l'autre). Comme on l'a dit précédemment, comme la table est générée aléatoirement, il y a des doublons et souvent les chaines se rejoignent et donc avec 1600 entrées dans la table * 100 mot de passe par chaine, on couvre une bonne partie des mots de passe (il y en a 1 000 000 en tout, soit 16%). Avec l'aléatoire, on ne peut de toute façon pas avoir une table exhaustive de mots de passe.

Implémentation de la table de Hashage

Pour une Rainbow table, une table de hash est un fait un tableau (dont la taille à été justifiée ci-dessus) de Noeud contenant P0 et P999, avec P999 comme clé (c'est sur P999 que l'on fera des recherche par la suite dans la table).
On défini l'objet `Noeud` de la manière suivante :

```
public class Noeud {
    private int P999;
    private int PX;
    // Getter et Setter
}
```

Maintenant pour l'objet table :

- On défini sa taille à l'instentiation
- On initialise toute ces cases à -1
- Ici on ne supprime pas d'élément, il n'y a donc pas d'élément DELETED, ce qui rend la table beaucoup plus simple.

Pour l'insertion, on va donc chercher une case de libre. Pour cela **on utilise la même fonction de hashage qu'au TP précédent**. On commence avec l'indice 0 (car c'est la première tentative d'insertion), et il y a plusieurs cas :

- Si on trouve une case vide (clé = -1), on sort de la boucle et on insère l'élément ici.
- Si un élément est déjà présent et qu'il à la même clé (P999), on ne fait pas insertion
- Si on est arrivé au bout de la table, on ne réalise pas l'insertion
- Sinon c'est que la case est occupé par un autre élément, on passe au sondage suivant (i++)

```
public boolean insert(int p999, int px) {
    int indice, i=0;
    do {
        indice = h(p999, i++);
        if (table[indice].getP999() == p999 || i>=table.length)
            return false;
    } while (table[indice].getP999() != -1);

    table[indice].setP999(p999);
    table[indice].setPX(px);
    return true;
}
```

Remplissage de la table de Hashage

Afin de remplir notre table, on utilise un `for` avec 10000 éléments. A chaque tour de boucle :

- On génére un mot de passe aléatoire sur 6 chiffres avec un objet `Random`
- On calcule la chaine pour obtenir P999

- On insère l'élément dans la table de hash

```
HashTable table = new HashTable(10061);

int nbitems=0;
for (int i=1; i<=10000; i++) {
    int px = rand.nextInt(1000000);
    int p999 = chaine.CalculChaine(0, px);

    if (table.insert(p999, px))
        nbitems++;
}
```

Sauvegarde dans un fichier

Afin de sauvegarder la table de Hashage, on se sert des fonctionnalité de Java : La sérialisation et en particulier `ObjectOutputStream`. Cela revient à stocker l'objet dans un fichier. On pourra alors restaurer cet objet dans un autre programme afin de réutiliser cette table (et les méthodes qui lui sont associées)

```
File file = new File("rainbow.table");
ObjectOutputStream ecriture;
try {
    ecriture = new ObjectOutputStream( new FileOutputStream(file));
    ecriture.writeObject(table);
       ecriture.close();
} catch (IOException e) {
    System.out.println("Erreur d'écriture");
    e.printStackTrace();
}
```

Exécution du code

L'exécution de ce programme est assez rapide puisqu'il faut en moyenne **15 secondes** pour calculer la table et la sauvegarder dans un fichier. On obtient alors un fichier binaire qui contient la table de hashage.
Notons qu'ici c'est assez rapide étant donné que la table est finalement construite pour des mots de passes peu nombreux et assez cours. Dans la réalité, le calcul d'un table peut être vraiment long suivant le nombre de mot de passe à crypter… D'où l'avantage de la calculer une bonne fois pour toute !

Exercice 4 : Utilisation de la table pour casser un mot de passe

Chargement de la table

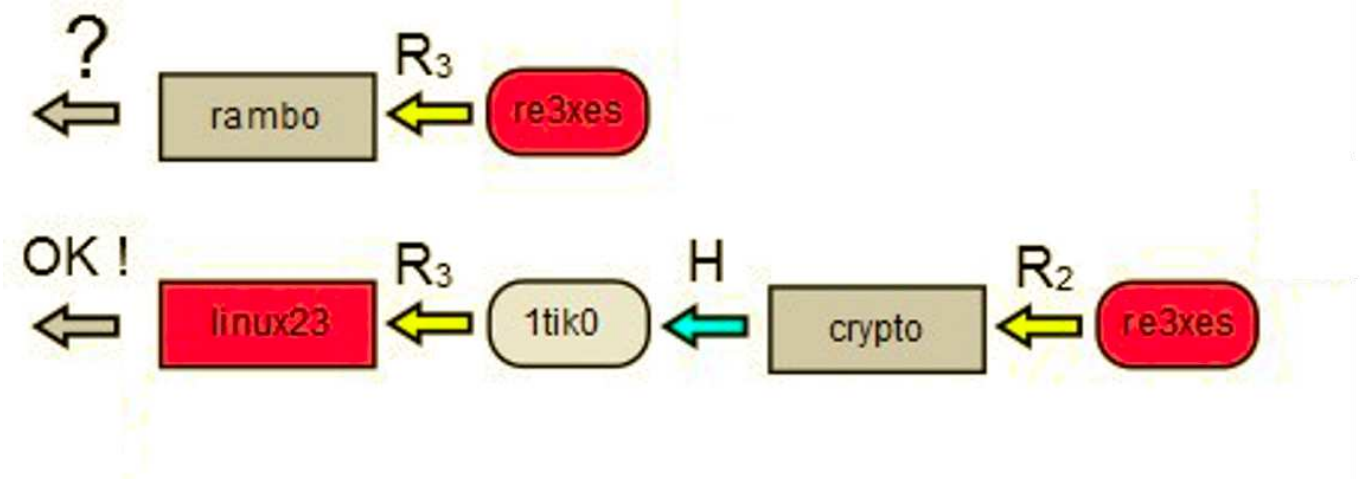
Comme la table est dans un fichier, on va l'importer afin de pouvoir s'en servir pour casser les mots de passe.
Tout d'abord on réimporte l'objet Hashtable avec `ObjectInputStream` :

```
File file = new File("rainbow.table");
ObjectInputStream lecture;
try {
    lecture = new ObjectInputStream( new FileInputStream(file));
    table = (HashTable) lecture.readObject();
       lecture.close();
} catch (IOException | ClassNotFoundException e) {
    System.out.println("Erreur importation");
    e.printStackTrace();
}
```

Utilisation de la Rainbow Table

Afin de décrypter le mot de passe à partir du Hash, nous allons faire comme si le Hash était un Hash d'une étape de construction de la chaine. Au début, on va considérer que c'est le Hash 999, il ne reste donc qu'à le réduire pour avoir P999 . A chaque fois, on vérifie si ce P999 est dans notre Table ou non. Si ce n'est pas le cas, on recommance, mais une étape plus tôt : on considère que le Hash est le 998, il faut donc lui appliquer R998 puis H puis R999 pour avoir le P999, que l'on va chercher dans notre table. Etc.

Voici un exemple si l'on cherche la chaine `re3xes` sachant que le nombre de réduction maximale est 3 :

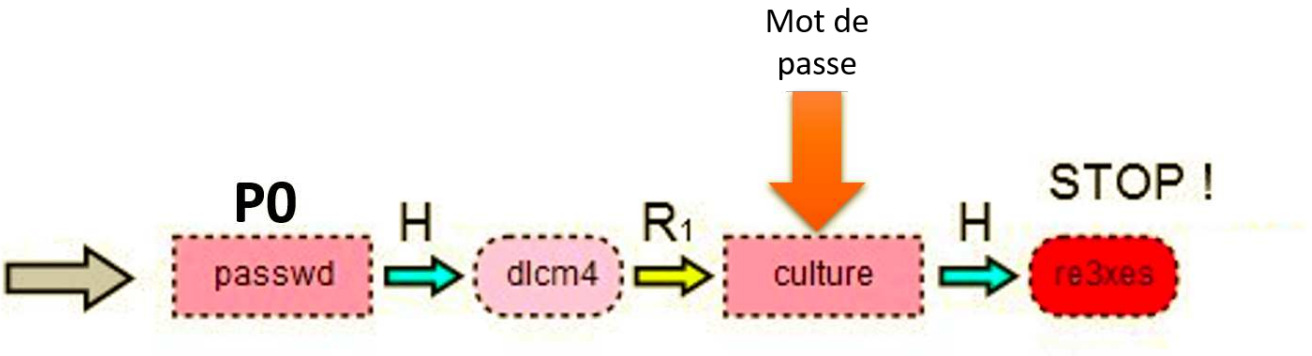


Afin de vérifier l'existence d'un P999 et de pouvoir récupère le mot de passe d'origine (P0) dans la table, nous avons défini une méthode qui prend en paramètre la clé recherché et retourne le P0 de cette clé s'il existe, en le cherchant dans la hash table. Le principe est de calculer, grâce à la fonction de hashage l'indice auquel doit se trouver l'élément, et s'il ne s'y trouve pas réessaie avec le numéro de sondage suivant. Si n'est pas présent (case vide), on retourne -1.

```
public int acces (int cle) {
    int indice, i=0;
    do {
        indice = h(cle, i++);
        if (table[indice].getP999() == cle)
            return table[indice].getPX();
    } while (table[indice].getP999() != -1 && i<table.length);
    return -1;
}
```

A partir du moment où l'on est retombé sur un P999 qui l'on connait, on se trouve dans l'étape 2.
On récupère le P0 associé, on repart de ce P0 et on lui applique successivement le Hash et la fonction de réduction :

- Si on retombe sur le Hash qui est passé en paramètre, alors le mot de passe est la fonction réduite avant ce Hash
- Si on arrive à P999 sans avoir trouvé le Hash, c'est une fausse alerte, et on reprend à l'étape 1 !



Afin de comparer le hash fourni en paramètre et ceux calculé dans la Hash Table, nous avions tout d'abord utilisé la méthode `.equal` pensant que celle-ci allait faire la comparaison des éléments. En réalité, ce n'était pas le cas, car elle compare uniquement l'identité des objets (et non pas leur égalité). Nous avons donc défini une méthode pour faire cela :

```
public static boolean compare(byte[] a, byte[] b) {
    for(int i=0;i<a.length;i++) {
        if(a[i] != b[i])
            return false;
    }
    return true;
}
```

Dans l'implémentation, nous récupèrons le Hash a décrypter en argument du programme. Une des difficultés est de correctement convertir cette String en tableau d'octet. Pour cela nous utilisons la méthode `parseHexBinary` . On retrouve dans le code, les 2 parties expliquées ci-dessus.
Voici le code de l'algorithme :

```
System.out.println("Décodage du mot de passe ...");
        int P0, px = 0;
        int i = 999;
        byte[] mdp = DatatypeConverter.parseHexBinary(args[0]);
        boolean find = false;

        while (!find) {
            //Partie 1
            do {
                px = chaine.reduction(mdp, i);
                px = chaine.CalculChaine(i + 1, px);
                i--;
            } while ((P0 = table.acces(px)) == -1 && i >= 0);

            if (i < 0) {
                System.out.println("Mot de passe non trouvé :-( ");
                System.exit(-1);
            }

            //Partie 2
            int j = 0;
            byte[] hash = chaine.md5(P0);
            while (!compare(hash, mdp) && j <= 1000) {
                px = chaine.reduction(hash, j++);
                hash = chaine.md5(px);
            }

            if (j <= 1000)
                find = true;
        }
        System.out.println("Mot de passe : " + px);
```

Test de fonctionnement

Maintenant, si on essaie de décoder tout les mots de passe, on va bien sur retrouver les mêmes résultats. Voici les traces pour le décodage du mot de passe d'Alice :

```
MD5? 84E6A804E2069365DF19AB2D0157E818
Importation du fichier ...
Décodage du mot de passe ...
Mot de passe : 712333
```

Le résultat apparait alors TRÈS rapidement : **En moins d'une seconde on trouve le mot de passe décodé** ! On a même l'impression que le plus long n'est pas le décodage mais l'importation de la table. La Rainbow Table fonctionne donc correctement et comme prévu, les résultats sont assez rapide !

On recommence pour les mots de passes à 9 caractères

Le fonctionnement du programme principal reste la même chose, cependant quelques changement s sont à noter :

- Au niveau de la fonction de `CalculChaine` :
  - Le modulo de la fonction de réduction doit être de 1 000 000 000 : `res = res % 1000000000;`
  - Quand on converti le int en String, avec le Hash, on doit le faire sur 9 chiffres : `String.format("%09d", px)`

- Au niveau de la construction de la table :
  Comme on a plus de mot de passe, il faut augmenter la taille de la table et le nombre de valeur généré. Le problème c'est que si on prend un table de plus de 10 000 000 nœuds (ce qu'il faudrait pour avoir plus maximum de mots de passe), les temps de calcul sont beaucoup trop long -> Près de 10 min pour calculer seulement 1% de la table…. Du coup, nous avons un peu limité la table de la table :
  - Taille de la table : 1000003 (Nous avons enlevé un facteur 10 et pris un nombre premier)
  - Nombre de nœuds générés : 1000000

- Au niveau de l'utilisation de la table :
  - Aucun changement

Différences de performances :
Comme on a augmenté la taille de la table et le nombre de mot de passe possible, la durée de génération de la table est passée à environ 35 min.
Si maintenant on crypte le mot de passe 658974235 on trouve : aa4fe43fd7b4242820cca1776b89fb39.
On essaie de décrypter ce mot de passe :

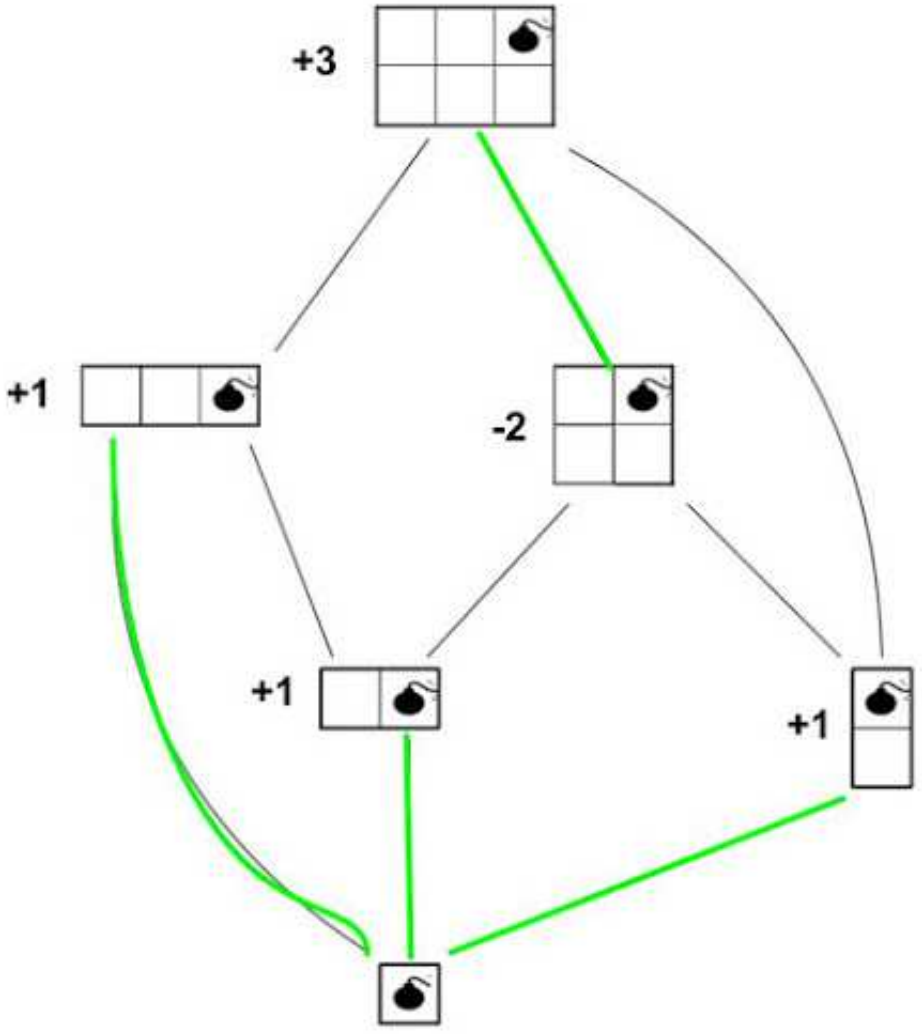
```
Importation du fichier ...
Décodage du mot de passe ...
Mot de passe : 658974235
```

Le traitement dur environ 10 sec : 7 sec pour le chargement de la table et 3 sec les calculs.
Notons tout de même que comme nous avons limité la taille de la table, il y a pas mal de mot de passe que nous ne sommes pas en mesure de décoder (par exemple : 486248625).
Il faudrait laisser tourner le programme plus longtemps et générer plus de valeur, pour avoir une meilleure probabilité de décoder tout les mots de passe.

TP7-8 : Programmation dynamique

Dans ce TP, nous allons implémenter algorithme permettant de calculer la valeur des positions dans le jeu de la tablette de chocolat et du carré de la mort. Pour cela nous allons faire différentes version de l'algorithme (récursif, dynamique) qui vont donc nous permettre de déterminer quel est le meilleur coup à jouer. Pour finir, nous pourrons alors faire une intelligence artificielle qui pourra jouer au jeu, sans jamais faire d'erreur.

Déterminer la valeur des positions suivantes



Pour calculer les valeurs, on part du carré de la mort. Toutes les combinaisons qui précédent le carré de la mort sont gagnante en un coup pour le joueur, car qu'il pourra donner le carré à l'adversaire. Ensuite on remonte de la même manière l'arbre des possibilité et à chaque fois on regarde la valeur de tout les successeurs d'une position pour en connaitre sa valeur.

Donner une formule mathématique pour calculer la valeur d'une position

Pour calculer la valeur d'une position, on regarde l'ensemble des successeurs :

- Si $(m, n) = (i, j)$, Alors $valeur = 0$
  On est dans le cas où il ne reste que la bombe, on a donc perdu !
- Si $\forall x \in successeurs, x > 0$, Alors $valeur = -(1 + max(successeurs))$
  C'est à dire que toutes les position sont gagnantes pour l'adversaire, et donc que l'on va choisir la position qui va le plus le ralentir.
- Si $\exists x \in successeurs, x \leq 0$, Alors $valeur = 1 - max(successeurs_{négatif})$
  Il existe au moins une solution perdante pour l'adversaire, on va donc prendre la solution qui le fait perdre le plus vite (ou qui termine la partie (x=0)).

Version Naïve du calcul de la valeur d'une position

Détail de l'algorithme

Le But de l'algorithme est de déterminer à partir du jeu actuel toutes les possibilités de jeu qui s'offrent à nous à cet instant précis (transformations du plateau de jeu) et d'en déduire la valeur du jeu actuel à partir de la valeur des plateaux suivants possibles.

Pour cela nous parcourons les coupes verticales possibles, puis toutes les coupes horizontales. Deux boucles `for` sont donc utilisées pour cette opération.

Durant cette étape de parcours,

- on cherche le plateau de jeu avec la valeur la plus élevée tant que toutes les valeurs sont positives (l'adversaire à un jeu gagnant mais on cherche à faire durer la partie).
- Si une des valeurs est négative, on cherche la valeur la plus élevée des négatives (au prochain coup l'adversaire à un plateau perdant mais on veut vite en finir).

L'algorithme est récursif, le cas de base est quand m = n = 1, le plateau vaut 0. Sinon on rappel la fonction avec un nouveau plateau correspondant à la tranche à tester.

Performances

Si on réalise un essaie avec la configuration présentée si-dessus : `f(3,2,2,0) = 3`
On retrouve bien le résultat donné ci-dessus. La durée de calcul est inférieure à la seconde.

Si on calcul : `f(10,7,7,3) = 11`
Ici le temps de calcul a bien augmenté puisque que l'on met 58 secondes à avoir le résultat.

Si on calcul : `f(10,7,5,3) = 15`
Le temps de calcul a légèrement augmenté : 73 secondes.

Complexité

Comme on peut le voir, la durée augmente très rapidement, ce qui est du à une complexité factorielle : On doit dès le début calculer m+n coupes (toutes les coupes possibles horizontale et verticale), et pour chacune de ces coupes on aura dans le *pire des cas* m+n-1 coupes à calculer, …. On a donc (m+n)*(m+n-1)*… Ce qui nous donne O((m+n)!)

On s'aperçoit, que dans la récursivité, on calcule plusieurs fois les mêmes configurations, ce qui prend bien entendu beaucoup de temps inutilement. On va donc essayer de garder les résultats déjà calculé en mémoire afin de pouvoir y accéder à nouveau facilement, sans devoir tout recalculer.

Code

```
public int f_rec(int m,int n,int i,int j) {
    if (m==1 && n==1) return 0;
    int maxSucc;
    if(m>=2){
        if(1<=i) maxSucc = f_rec(m-1,n,i-1,j);
        else     maxSucc = f_rec(1,n,i,j);
    }
    else {
        if(1<=j) maxSucc = f_rec(m,n-1,i,j-1);
        else     maxSucc = f_rec(m,1,i,j);
    }
    int maxSuccNegatif=42;
    boolean allSuccStrictPositif = true;
    int temp;
    if(m>=2){
        for(int k=1;k<m;k++) {
            if(k<=i) temp = f_rec(m-k,n,i-k,j);
            else     temp = f_rec(k,n,i,j)
            maxSucc = Math.max(temp, maxSucc);
            if(temp<=0 && allSuccStrictPositif) {
                allSuccStrictPositif = false;
                maxSuccNegatif = temp;
            }
            if(!allSuccStrictPositif)
                if(temp <= 0)
                    maxSuccNegatif = Math.max(temp,maxSuccNegatif);
        }
    }
    if(n>=2){
        for(int k=1;k<n;k++) {
            if(k<=j) temp = f_rec(m,n-k,i,j-k);
```

```
                else     temp = f_rec(m,k,i,j);
                maxSucc = Math.max(temp, maxSucc);
                if(temp<=0 && allSuccStrictPositif) {
                    allSuccStrictPositif = false;
                    maxSuccNegatif = temp;
                }
                if(!allSuccStrictPositif)
                    if(temp <= 0)
                        maxSuccNegatif = Math.max(temp,maxSuccNegatif);
            }
        }
        if(allSuccStrictPositif) return -(1+maxSucc);
        else return 1 - maxSuccNegatif;
    }
```

Version dynamique

Détail de l'algorithme

La version dynamique de l'algorithme utilise un tableau de cache pour stocker la valeur des plateaux déjà calculés. L'algorithme reste exactement le même.
Le cache est un tableau d'entier à 4 dimensions propre à l'instance de l'objet, cela permet de le réutiliser entre plusieurs appels de la fonction.
La taille du cache est:

- max(m,n) pour les 2 premières dimensions de m et n (pour stocker les symétries pour la version accélérée, ici des tailles de respectivement m et n auraient suffit)
- max(i,j) pour les 2 dimensions restantes pour les mêmes raisons

L'initialisation se fait dans le constructeur de l'objet à 0.

Ensuite, lors de l'appel de `f_dyn()`, on cherche un résultat dans le cache (une valeur de 0 indique qu'il faut calculer le résultat)

Une fois calculé, on place la valeur dans le cache et la retourne.
Encore une fois, tout est récursif.

```
    public int f_dyn(int m,int n,int i,int j) {
        if (tab[m][n][i][j] != 0)
                    return tab[m][n][i][j];

        // Calcul du résultat de la même manière que précédemment avec
    l'algorithme dynamique

        tab[m][n][i][j] = result;
                return result;
```

```
}
```

Performances

Pour tout les calculs qui ont été fait avec l'algorithme récursif ci-dessus, **le résultat est donné en moins d'une seconde** !

En effet, la complexité est maintenant en O(m*n) car on ne recalcule plus la même chose plusieurs fois.

Maintenant si on teste `f(100,100,50,50) = -198` , on met 17 secondes pour avoir le résultats.
Il nous faut aussi 15 secondes pour calculer : `f(100,100,48,52) = 191`

Cependant, à cause du tableau de cache, le programme prend maintenant plus de 1,5Go de mémoire vivre !

Trouver les paires (i, j) telles que f(127, 127, i, j) = 127

Afin de déterminer toutes ces paires, on calcul pour toutes les valeurs de i et j la valeur de la position et on la compare à 127 :

```
for (i=0; i<m; i++)
    for(j=0; j<n; j++)
        if (naif.f_dyn(m,n,i,j) == 127)
            System.out.println("("+i+","+j+")");
```

On obtient avec les couples suivant en 604 secondes :

```
(0,63), (63,0), (63,126), (126,63)
```

On obtient alors 4 couples (i,j). Remarquons que pour chaque couple, on a aussi son symétrique.

Version accélérée

Expliquer pourquoi toutes ces simulations ont la même valeur

Ces 8 plateaux ont la même valeur car ce sont des axes de symétrie du plateau : il y a un axe vertical, un horizontal et un diagonal soit 2^3 = 8 possibilités. Ainsi dans ces différents cas, les coupes sont identiques et donc la valeur du plateau est identique.

Détail de l'algorithme

La version accélérée est quasi identique à la version précédente, le changement se situe au niveau de l'écriture dans le cache.
Lors de l'écriture dans le cache d'une valeur pour un certain plateau, **on va directement écrire les 7 autres plateaux symétriques qui ont la même valeur.** De cette manière, nous n'aurons pas besoin de les calculer si l'on s'en sert.

```
private void cacheWriteAccel(int m,int n,int i,int j,int value) {
    tab[m][n][i][j] = value;
    tab[m][n][m-i-1][j] = value;
    tab[m][n][i][n-j-1] = value;
    tab[m][n][m-i-1][n-j-1] = value;
    tab[n][m][j][i] = value;
    tab[n][m][n-j-1][i] = value;
    tab[n][m][j][m-i-1] = value;
    tab[n][m][n-j-1][m-i-1] = value;
    }
```

Cette implémentation force à utiliser un cache carré pour les paires de dimensions 1,2 et 3,4. le cache est donc légèrement plus grand, mais la recherche y est instantanée.

Une autre approche aurait été de stocker dans le cache uniquement les valeur calculées (comme avant) mais d'effectuer un sondage sur les 8 variantes d'un plateau dans le cache avant de calculer la valeur.

Performance

Comme a chaque calcul, on détermine en réalité 8 valeurs, le gain devrait être de l'ordre de 8 en terme de temps de calcul.
Cependant comme on ne se sert pas de toutes ces valeurs, en réalité le gain n'est que de 6 (ce qui est déjà pas mal).

Si on essaie à nouveau de *Trouver les paires (i, j) telles que f(127, 127, i, j) = 127*, on obtient le même résultat que précédemment, en seulement * *45 sec**. Ici on a même divisé le temps par 12 (on doit gagner aussi du temps de calcul).

Comparaisons : Bilan

Durée (sec)	Récursif	Dynamique	Avec symétrie
f(3,2,2,0)	<1	<1	<1
f(10,7,7,3)	58	<1	<1
f(10,7,5,3)	73	<1	<1

f(100,100,50,50)	-	17	2.5
f(100,100,48,52)	-	15	2

Programmation du moteur de jeu

Afin d'utiliser nos fonctions qui calculent les valeurs d'une position, nous allons créer un jeu où le joueur pourra affronter l'ordinateur (qui jouera du mieux possible étant donné qu'il calcul tout les coups).

Pour cela, on utilise un objet `Plateau` qui sera défini par les entiers m, n, i, j. Afin de savoir si la partie est déterminée, on regarde si i=j=0 et m=n=1 signifiant que l'on se trouve dans la position où il ne reste que la bombe. Ensuite, on alterne les coups où l'algorithme joue et ou le joueur joue :

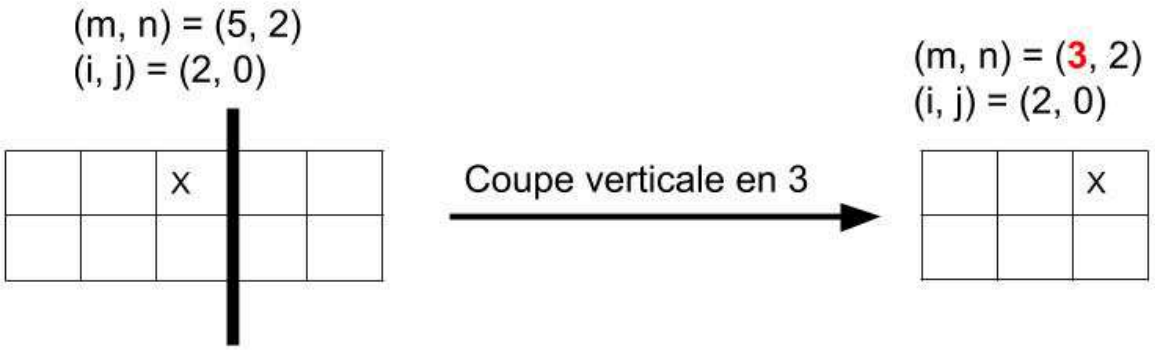
- Quand c'est un tour du pc
    - Avec une procédure similaire à celle utilisée précédemment, on calcul quel est le meilleur coup à jouer. La différence est dans le fait que l'on ne retourne pas uniquement le meilleur coup à jouer, mais aussi quelle est la coupe à faire pour avoir le meilleur coup. Pour cela, on utilise un plateau temporaire qui contient la meilleure coupe que l'on puisse faire.

- Quand c'est au tour du joueur :
    - il entre le sens de la coupure (horizontale ou vertical)
    - il entre l'indice de la coupure
    - Si les informations entrées sont correctes, on applique les modifications sur le plateau de jeu (*voir détails en dessous*)
    - Sinon on lui demande de jouer à nouveau

A chaque tour, on affiche la valeur du plateau actuel, et le plateau de jeu avec des `O` pour les emplacements vite et `X` pour la bombe.

Pour modifier le plateau de jeu :
Il faut tout d'abord distinguer les deux sens de coupe : horizontal et vertical.
Ensuite pour chaque sens, il faut distinguer les cas suivants :

- **Si la bombe est avant la coupe** : il suffit donc de réduire le plateau de jeu (modifier m ou n suivant le sens)
- **Si la bombe est après la coupe** : il faut bien entendu réduire le plateau de jeu mais aussi décaler la bombe suivant la taille de la coupe.




```
public boolean coupe (Sens sens, int a) {
    if (sens == Sens.horizontal) {
        if (0<a && a<m && a<=i){
            m = m-a;   i = i-a;
            return true;
        }
        else if (0<a && a<m && a>i){
            m = a;
            return true;
        }
    }
    else if (sens == Sens.vertical) {
        if (0<a && a<n && a<=j){
            n = a;   j = j-a;
            return true;
        }
        else if (0<a && a<n && a>j){
            n = a;
            return true;
        }
    }
    System.out.println("Coordonnée invalide");
    return false;

}
```

Capture d'une partie de jeu

Jeu avec m=3, n=2, i=2, j=0 :

```
Representation du jeu :
O O X
O O O

----- Au tour du PC -----
Evalution de la configuration : 3
Representation du jeu :
O X
O O

----- A vous de jouer ! -----
Evalution de la configuration : -2
Taper le sens de la coupure : h/v
v
Taper l'indice de la coupure :
1
Representation du jeu :
X
O

----- Au tour du PC -----
Evalution de la configuration : 1
Representation du jeu :
X

Vous avez perdu face à l'ordinateur !
```

Jeu avec m=10, n=7, i=7, j=1 :

```
Representation du jeu :
O O O O O O O O O O
O O O O O O O O O O
O O O O O O O O O O
O O O O O O X O O
O O O O O O O O O O
O O O O O O O O O O
O O O O O O O O O O

----- Au tour du PC -----
Evalution de la configuration : 11
Representation du jeu :
O O O O O
O O O O O
```

```
O O O O O
O O X O O
O O O O O
O O O O O
O O O O O

----- A vous de jouer ! -----
Evalution de la configuration : -10
Taper le sens de la coupure : h/v
h
Taper l'indice de la coupure :
3
Representation du jeu :
O O X O O
O O O O O
O O O O O

----- Au tour du PC -----
Evalution de la configuration : 5
Representation du jeu :
X O O
O O O
O O O

----- A vous de jouer ! -----
Evalution de la configuration : -4
Taper le sens de la coupure : h/v
v
Taper l'indice de la coupure :
1
Representation du jeu :
X
O
O

----- Au tour du PC -----
Evalution de la configuration : 1
Representation du jeu :
X

Vous avez perdu face à l'ordinateur !
```