TD4 : Le problème de la pyramide

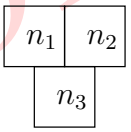
TOP: Techniques and tOols for Programming

Première année

L'objectif de ce TD et du TP associé est de réaliser un algorithme de recherche avec retour arrière pour résoudre ce qu'on appelle le problème de la pyramide.

Présentation du problème On considère une pyramide la tête en bas de hauteur n comme celle représentée plus bas. On cherche à remplir toutes les cases avec chacun des entiers compris entre 1 et $\frac{n(n+1)}{2}$ en respectant les contraintes suivantes :

1. Chaque nombre de $\left[1, \frac{n(n+1)}{2}\right]$ ne figure qu'une fois sur la pyramide
2. La valeur de chaque case est égale à la différence des deux cases placées au dessus d'elle. Ainsi sur la figure ci-contre, $n_3 = |n_1 - n_2|$



Certaines instances du problème (certains n) n'admettent pas de solution (cf. dernier exercice).

★ Exercice 1: Représentation mémoire

La première idée pour représenter la pyramide est d'utiliser la moitié d'un tableau à deux dimensions (`int pyr[ ][ ]`), mais l'autre moitié serait réservée pour rien.

Pour économiser la mémoire, nous allons utiliser un tableau à une dimension en rangeant les différentes «tranches de pyramides» les unes à coté des autres. Selon la façon de couper ces tranches, il existe de nombreuses manières de numéroter les cases. Les figures 2 et 3 présentent deux numérotations différentes.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

FIG. 1 – Représentation mémoire.

10	11	12	13	14
6	7	8	9	
3	4	5		
1	2			
0				

FIG. 2 – Une numérotation par lignes.

0	1	3	6	10
2	4	7	11	
5	8	12		
9	13			
14				

FIG. 3 – Une numérotation par colonnes.

▷ **Question 1:** Écrivez une fonction `indiceLigne(ligne,colonne)` calculant l'indice de la case placée sur la `ligne` et sur la `colonne` indiquée en suivant la numérotation par lignes représentée sur la figure 2. INDICATION : calculez le nombre de case dans la pyramide de taille `ligne`.
`indiceLigne(1,1)=0`; `indiceLigne(2,2)=2`; `indiceLigne(3,2)=4`; `indiceLigne(4,2)=7`.

Réponse

```
private int indiceLigne(int lig, int diag){
  return (lig * (lig - 1 ) / 2 + diag - 1);
}
```

Fin réponse

▷ **Question 2:** Écrivez `indiceColonne(ligne,col)` utilisant la numérotation par colonnes de la figure 3.
`indiceColonne(1,1)=0`; `indiceColonne(2,2)=2`; `indiceColonne(2,3)=4`; `indiceColonne(2,4)=7`.

Réponse

```
//@requires lig >= 1 && lig <= hauteur() && diag >= lig && diag <= hauteur();
private  /*@pure@*/ int indiceColonne(int lig, int diag) {
  return (diag * (diag - 1 ) / 2 + lig - 1);
}
```

Faut au moins qu'ils trouvent la deuxième depuis la première semble tout à fait possible.

Fin réponse

★ **Exercice 2: Algorithme «générer puis tester« (première approche)**

La première idée est de générer toutes les pyramides existantes, puis de vérifier à postériori si elles vérifient les contraintes ou non. Il faut donc générer toutes les permutations de la liste des n premiers entiers puis chercher celles vérifiant la seconde contrainte (puisque la première est vérifiée par construction).

▷ **Question 1:** Donnez un algorithme permettant de calculer les permutations des n premiers entiers.

Réponse

Récursif, et pour chaque position, on prend tous les nombres pas encore pris. Autrement dit, c'est du back-tracking... C'est une bonne version simplifiée de ce qui vient après. Pour faire propre, le mieux est de sortir un itérateur. Mais comme c'est un peu pénible de couper la récursion au milieu et de se souvenir de où on est, le plus simple est de générer brutalement toutes les permutations à parcourir d'abord, puis d'itérer bien gentiment dessus. C'est brutal, ca a des perfs de daube, mais à ce point, on s'en tape. Et en plus, pour l'itération, on fait de la délégation (ie, on réutilise Iterator de ArrayList sans réfléchir).

```
1  public class PermutIter implements Iterator {
2    private ArrayList<int[ ]> elems;
3    private Iterator<int[ ]> it;
4
5    public PermutIter(int taille) {
6      elems = new ArrayList<int[ ]>();
7      genere(new int[taille], 0);
8      it = elms.iterator();
9    }
10   public int[ ] next() {
11     return it.next();
12   }
13   public boolean hasNext() {
14     return it.hasNext();
15   }
16   private genere(int [ ] tab, int rang) {
17     // cas de base
18     if (rang == tab.length) {
19       elems.add(tab.clone());
20       return;
21     }
22     // cas général
23     for (int n=0; n<tab.length; n++) {
24       if (! contains(tab, n, rang)) {
25         tab[rang] = n;
26         genere(tab, rang+1);
27       }
28     }
29   }
30   private boolean contains(int[ ] tab, int value, int rang) {
31     // trivial, right? On ne regarde que dnas les rang premières cases
32   }
33 }
```

Pour leur faire trouver, faut faire à la main ("parametre de récursion", "cas de base", etc). Mais ca suffit pas, faut les convaincre que ca fonctionne. Pour aider, prendre le cas pour 3 : ca génère (dans l'ordre) 012, 021, 102, 120, 201, 210. Faut faire sentir les appels récursifs. Dessiner un arbre des appels peut aider, si vous en sentez le besoin.

Fin réponse

▷ **Question 2:** Écrivez la fonction `correcte()` qui teste récursivement si la pyramide est valide.

Réponse

Note : C'est un bon moment pour rappeler qu'en Java, on peut avoir plusieurs fonctions de même nom, mais de profils différents.

De plus, les tests sur contains sont inutiles ici (puisque c'est vrai par construction), mais c'est la fonction qu'il faudra écrire plus tard pour faire fonctionner propage()

```
public /*@ pure @*/ boolean correcte(){
    return correcte(hauteur);
}

 /* Renvoi vrai si la pyramide est correctement construite jusqu'à la diagonale diag */
 //@ requires diag >= 0 && diag <= hauteur() ;
 //@ ensures correcte() ;
 private  /*@pure@*/ boolean correcte(int diag){
     // Récursivité portant sur les diagonales de 1 à diag

     if (diag == 0)
       return true;

     for (int lig=1; lig<=diag; lig++)
       if (posCorrecte(lig,diag) == false)
         return false;

     return correcte(diag-1);
 } // correcte(int)


 /* Renvoi vrai si la position pointée est correcte */
 //@ requires lig >= 1&& lig <= diag+1
 //@          && diag >= 1 && diag <= hauteur() ;
 public  /*@pure@*/  boolean posCorrecte(int lig, int diag) {
     int nbre = valueAt(lig,diag);

     /* jamais le droit de sortir de [1..count] */
     if (!nombreAutorise(nbre))
         return false;

     /* Aucune [autre] contrainte sur la premiere case */
     if (lig == 1 && diag == 1)
         return true;

     /* il faut toujours être original */
     if (contains(nbre,lig-1,diag))
         return false;

     /* Il faut être la différence des ancêtres (sauf sur la première ligne) */
     if (lig > 1) {
         int n1 = valueAt(lig-1,diag-1);
         int n2 = valueAt(lig-1,diag);

         if (nbre != Math.abs(n1 - n2) )
             return false;
     }

     /* Tous les tests ont été passés avec succès */
     return true;
 } // diagonaleCorrecte(int, int)

 /* Renvoie vrai si val est dans la pyramide avant la position (lig,diag) */
 //@ requires lig >= 1 && lig <= hauteur()
 //@          && diag >= 1 && diag <= hauteur();
 private  /*@pure@*/ boolean contains(int val, int lig, int diag) {
     /* On tient compte de l'implantation de la pyramide par soucis d'optimisation */
```


CORRECTION

```
        int fin = indice(lig,diag);
        for (int i=0; i < fin; i++) {

            if (val == elements[i])
                return true;
        }

        return false;
    }//contains

    /* Renvoi vrai si le nombre val est dans l'intervalle [ 1 .. count] */
    private  /*@pure@*/ boolean nombreAutorise(int v){
        return (v >= 1 && v <= count());
    }
}
```

Fin réponse

▷ **Question 3:** Dénombrez le nombre d'opérations que cet algorithme réalise.

Réponse

- Il y a $n!$ listes à construire
- Pour chacune, le processus de test est de complexité $O(n)$ car il y a n cases à tester donc, le gros if au milieu sera appellé n fois sur l'ensemble des appels.

La complexité est donc en $O(n \times n!)$, ce qui est **énorme**.

Fin réponse

★ **Exercice 3: Algorithme de construction pas à pas (deuxième approche)**

La solution de l'exercice précédent est inefficace car elle construit toutes les solutions possibles, même celles ne respectant pas les contraintes du problème. Une amélioration possible consiste à vérifier à chaque étape de la construction que ces contraintes sont respectée, et à s'interrompre dès qu'un choix mène à une situation interdite.

Il s'agit donc d'un algorithme récursif avec retour arrière. Pour découvrir le paramètre d'induction, il faut s'interroger sur la façon de couper une pyramide en une pyramide de plus petite taille. La première façon est celle dite «en ligne» (comme sur la figure 4), on ajoute des lignes plus longues. Le paramètre d'induction est alors le numéro de ligne en cours de remplissage. Une autre façon est d'agir «en colonnes» (comme sur la figure 5) on ajoute des «tranches» de la pyramide. Le paramètre d'induction est alors la diagonale en cours de traitement.

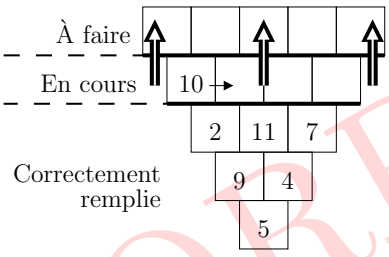


FIG. 4 – Remplissage partiel par lignes.

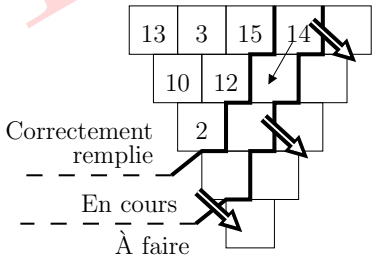


FIG. 5 – Remplissage partiel par colonnes.

L'approche «en colonnes» est préférable car la seconde contrainte lie un nombre aux deux placés au dessus de lui. Il est donc naturel de chercher à traiter le nombre du dessous juste après un nombre donné. Cela permet de s'assurer que toute solution correcte aux étapes précédentes de la récursion ne gênera pas le respect de la seconde contrainte. Au contraire, il est possible que l'approche «en ligne» mène à une situation de blocage due à la seconde contrainte à l'étage n nécessitant de modifier les étages inférieurs.

▷ **Question 1:** Écrire le corps de la méthode `void remplir()`. Elle lance une fonction récursive dotée de plus de paramètres en initialisant les paramètres convenablement.

Réponse

```
//@ ensures correcte() ;
public void remplir(){
    remplir(1);  // on commence a la diagonale 1
}
```

Fin réponse

★ **Exercice 4: Écriture de la fonction récursive**

Après avoir trouvé le schéma général de la récursion, il nous faut écrire la fonction récursive elle-même.

Le paramètre de la récursion est la diagonale en cours de traitement ; la condition d'arrêt est le dépassement de la taille de la pyramide : si le paramètre `diag` dépasse la hauteur de la pyramide, le problème est résolu et la pyramide est complètement correctement résolue.

Si on parvient à remplir correctement la diagonale `diag`, il convient de réaliser un appel récursif avec `diag+1` pour tenter de remplir le reste de la pyramide. Sinon, on effectue un retour arrière.

On constate que l'on peut calculer (par une série de soustraction) la diagonale entière à partir de la solution partielle et du nombre placé sur la première ligne de de la diagonale. La fonction récursive `remplir(diag)` consiste donc à tester chaque entier en première position de la diagonale, puis à tenter de le «propager», *i.e.* à vérifier que la diagonale induite par ce nombre est valide. On échoue si on est amenés à réutiliser un nombre déjà utilisé dans la pyramide.

▷ **Question 1:** Écrire la méthode `boolean propage(int diag, int val)` qui tente de propager une valeur donnée sur une diagonale donnée.

Réponse

```
/*
 * si la propagation de val est possible sur la diagonale diag
 * modifie elements en conséquence et renvoie vrai.
 * si la propagation ne peut pas se faire, renvoie faux
 */
//@ requires diag >= 1 && diag <= hauteur() && val >= 1 && val <= count();
//@ ensures !\result  || (\result && diagonaleCorrecte(diag,diag)) ;
private boolean propager(int diag, int val){
    // On pose val en haut de la diagonale
    setValueAt(val,1,diag);

    // On essaie de propager sur le reste de la diagonale

    // Itération conditionnelle portant sur le domaine [ 2 .. diag ]
    // On s'arrête dès que les contraintes ne sont plus respectées

    int nbre = val;
    for (int lig=2; lig <= diag; lig++) {
        /** invariant
            -- la diagonale diag est <<bien construite>> jusqu'en lig-1
            diagonale_correcte(lig-1,diag)
        **/
        /** variant        diag - lig  **/
        nbre -= valueAt(lig-1,diag-1);    // économise les accès au tableau
        nbre = Math.abs(nbre);
        // On s'assure que le nombre ainsi calculé n'est pas déjà utilisé
        if (contains(nbre,lig,diag))
            return false;

        // Le nombre n'est pas utilisé; on le pose et on continue
        setValueAt(nbre,lig,diag);

    }
    return true;
}//propager(int,int)
```

Fin réponse

▷ **Question 2:** Écrire la méthode `boolean remplir(int diag)` qui tente de remplir la diagonale donnée.

Réponse

```
/* Remplir a partir de la diagonale diag */
//@ requires diag >= 1 && diag <= hauteur() + 1 && correcte(diag-1) ;
//@ ensures  (\result && correcte()) || !\result  ;
private boolean remplir(int diag){
    // Itération conditionnelle parcourt le domaine [count .. 1]
    // Arret lorsqu'une solution est trouvee

    if (diag > hauteur)
        return true;

    for (int k=count(); k>=1; k--) {

        // si k est déjà utilisé, on passe au tour de boucle suivant
        if (!contains(k,1,diag)) {

            // Essayer de propager k sur la diagonale

            if (propager(diag,k)) {
                // La propagation a été faite, on essaie de remplir la suite
                if (remplir(diag+1)) {
                    // L'appel récursif est une réussite : on tient une solution
                    return true;
                }
            }

        } // k n'est pas encore pris
    }
    return false;
}//remplir(int)
```

Fin réponse

★ **Exercice 5: Pour aller plus loin**

La figure ci-contre donne les temps de calcul (en secondes sur un centrino 1.5Ghz) et les taux de remplissage obtenus pour des pyramides de différentes tailles. La dernière ligne indique donc que la recherche des pyramides de taille 12 a duré plus de 28h, et que la meilleure solution ne remplit que 73% du tableau.

Rang	Remplissage	Temps
2	3/3	0,002
3	6/6	0,002
4	10/10	0,003
5	15/15	0,006
6	20/21	0,12
7	25/28	0,9
8	31/36	6
9	37/45	70
10	43/55	948
11	49/66	11551
12	57/78	103671

Il semble donc que ce problème n'admette pas de solution pour $n > 5$. On peut donc le généraliser de la façon suivante : on s'autorise à laisser de coté des nombres lors du remplissage de la pyramide. Il s'agit alors de minimiser le numéro M de la plus grande valeur utilisée dans une pyramide à n étages n'utilisant que des nombres entiers strictement positifs distincts tels que chaque valeur (sauf celles de la dernière ligne) soit la différence des deux valeurs situées immédiatement au-dessous.

Pour $n = 5$, on a alors $M = \frac{n(n+1)}{2} = 15$

Si vous trouvez une solution pour $n > 5$, merci de la communiquer à l'équipe enseignante (qui n'en connait pas).

Réponse

Ca, c'est juste pour occuper ceux qui savent déjà programmer. Dites leur de m'envoyer leurs solutions par email privé...

Fin réponse

CORRECTION