TDP4 : Récursivité (pyramide)

TOP : Techniques and tOols for Programming – 1A

L'objectif de ce TD et du TP associé est de découvrir la notion d'algorithme de recherche avec retour arrière (backtracking) au travers du problème classique de la pyramide.

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

1. Chaque nombre de $\left[1, \frac{h(h+1)}{2}\right]$ 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, $n_3 = |n_1 - n_2|$

n_1	n_2
n_3	

Ce problème se pose par exemple lorsque l'on cherche à disposer les boules de billards en respectant les contraintes données (dans ce cas, $h = 5$). Certaines instances du problème (certains h) n'admettent pas de solution (cf. dernier exercice) tandis que d'autres admettent plusieurs solutions (8 solutions pour $h = 3$).

★ **Exercice 1: Représentation mémoire** La première idée pour représenter la pyramide est d'utiliser la moitié d'un tableau à deux dimensions de type `Array[Array[Int]]`, mais une seule moitié serait utilisée et l'autre serait réservée pour rien.

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

FIGURE 1 – Représentation mémoire contiguë.

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. Nous allons pour instant prendre la plus simple, c'est à dire numéroter les cases par lignes comme sur la figure ci-contre.

Il nous faut définir une fonction `indiceLigne(ligne,colonne)` calculant l'indice de la case placée sur la `ligne` et sur la `colonne` indiquée en suivant cette numérotation. Notez que : `indiceLigne(1,1)=0`; `indiceLigne(2,2)=2`; `indiceLigne(3,2)=4`; `indiceLigne(4,2)=7`.

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

FIGURE 2 – Numérotation en ligne.

▷ **Question 1:** Écrivez cette fonction `indiceLigne(ligne,colonne)`, et explicitez ses préconditions. INDICATION : calculez le nombre de case dans la pyramide de hauteur `ligne`.

★ **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.

▷ **Question 2:** Écrivez la fonction `correcte()` qui teste si une permutation donnée forme une pyramide valide. Il suffit de vérifier que chaque élément est la différence de ceux placés au dessus, puisque toutes les valeurs sont présentes par construction. La fonction `indiceLigne()` est utile ici.

▷ **Question 3:** Sur machine, écrivez le code manquant pour trouver toutes les pyramides convenables de hauteur 3.

▷ **Question 4:** Dénombrez le nombre d'opérations que cet algorithme réalise.

★ **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 toutes les contraintes du problème. Une amélioration possible consiste donc 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. On appelle ce genre d'algorithmes des algorithmes récursifs avec retour arrière (*backtracking algorithms* en anglais).

▷ **Question 1:** Modifiez la fonction correcte précédemment écrite[1] afin qu'elle ne vérifie que le début

1. Lors du TP sur machine, vous devriez faire une copie de votre travail précédent afin de pouvoir comparer les versions.

du tableau, sans considérer les éléments placés après le paramètre `rang` qui ne sont pas initialisés.

▷ **Question 2:** Modifiez l'algorithme de génération des permutations écrit plus tôt afin de couper dès que la solution partiellement construite ne respecte pas la seconde contrainte du problème $n_1 = |n_2 - n_3|$

▷ **Question 3:** Sur machine, comparez le temps d'exécution de cette version avec celle de la version précédente pour la hauteur 3. Si on mesure (avec la fonction System.nanotime) le temps t_1 avant l'opération et le temps t_2 après coup, la durée de l'opération en secondes est $\frac{t_2 - t_1}{10^9}$.

★ Exercice 4: Génération par tranches (amélioration de l'approche)

Couper les branches menant à des solutions invalides de la sorte s'avère incroyablement plus rapide que la précédente. Cela permet de trouver des pyramides de hauteur 5 en quelques secondes. Mais pour aller plus loin, il faut raffiner cette approche.

Pour cela, l'objectif est de générer les contraintes le plus tôt possible, pour s'éviter de d'agrandir des solutions partielles vouées à l'échec. Par exemple, s'il s'avère impossible de placer une valeur dans la case 11 qui respecte les contraintes, des cases comme 8, 9, 4, 5 ou 2 auront été remplies pour rien. L'objectif est donc de changer l'ordre de remplissage pour détecter les blocages le plus tôt possible et trouver les solutions plus vite.

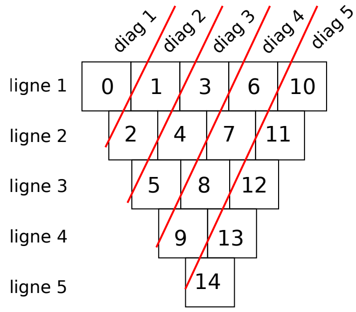

Figure 3 – Numérotation diagonale

L'approche «en colonnes» est préférable car la seconde contrainte lie un nombre aux deux places 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:** Plutôt que de réaliser un parcours compliqué, nous allons renuméroter les cases de la pyramide pour que l'ordre naturel expose les contraintes au plus tôt. Écrivez une méthode `indiceDiag()` semblable à `indiceLigne()` définie précédemment, mais numérotant les cases comme sur la figure 3.
`indiceDiag(1,1)=0;` `indiceDiag(2,2)=2;` `indiceDiag(2,3)=4;` `indiceDiag(2,4)=7.`

▷ **Question 2:** Écrivez une nouvelle méthode **correcte()** vérifiant que la pyramide respecte la seconde contrainte du problème dans le nouveau repère de numérotation.

▷ **Question 3:** Sur machine, comparez le temps d'exécution de cette nouvelle version par rapport à la précédente (il n'est pas nécessaire de modifier la fonction de génération des permutations pour cela).

★ Exercice 5: Génération par propagation (amélioration de l'amélioration)

Les résultats pratiques obtenus à l'exercice précédent sont décevants, et il faut encore affiner notre approche. Pour cela, on remarque qu'il est inutile de tester toutes les valeurs possibles pour n_3 puis ne garder que celles qui respectent les contraintes, car une fois que n_1 et n_2 sont connus, une seule valeur est possible pour n_3. L'idée est alors de placer une valeur possible en haut de la diagonale, puis de la propager en vérifiant qu'on respecte la première contrainte. La seconde sera respectée par construction.

▷ **Question 1:** Réécrivez l'algorithme sous forme d'une récursion portant sur la diagonale à remplir et non sur chaque case de la pyramide comme précédemment. À chaque étage de la récursion, il faut tester toutes les valeurs possibles à placer sur la première ligne, à tour de rôle. Il faut ensuite propager cette valeur en remplissant successivement les cases avec les valeurs imposées par la seconde contrainte. Si la valeur à placer est déjà utilisée ailleurs dans la pyramide, il faut couper court à la génération et explorer une autre branche. Si on parvient à remplir cette diagonale, il faut (tenter de) remplir la suite par un appel récursif sur la diagonale suivante. Vous aurez probablement besoin d'écrire les fonctions suivantes :

– `contient(pyr:Array[Int], valeur:Int, rang:Int)` qui retourne vrai si la `valeur` est présente dans le début de la `pyramide` (en ne considérant que les cases jusqu'à la position `rang`).
– `propage(pyr:Array[Int], value:Int, diagonale:Int):Boolean` qui tente de propager cette `valeur` sur cette `diagonale` par une série de soustractions.
– `genere(pyr:Array[Int], diagonale:Int)` qui tente de remplir récursivement la `pyramide` sachant que toutes les diagonales inférieures à `diagonale` sont déjà remplies correctement. La condition d'arrêt de cette récursion est quand la pyramide est intégralement remplie, ou quand aucune valeur possible ne peut être propagée correctement.

▷ **Question 2:** Sur machine, vérifiez que cette version retrouve toutes les solutions trouvées par les algorithmes précédents. Chronométrez cette version de votre code pour les hauteurs 5, 6 et 7. Ce problème n'admettant pas de solution pour $h = 6$ ni pour $h = 7$, il est normal que votre code n'en trouve pas.

★ **Exercice 6: 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).