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.

Réponse

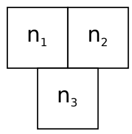
Pour plus de fun, l'amphi correspondant est la semaine prochaine. On appelle ça de l'enseignement inversé (ou un module mal préparé, au choix).

Plus sérieusement, il n'est absolument pas attendu que l'on finisse le TDP avec les élèves. Seuls les plus motivés commenceront le dernier exercice (mais faut bien donner du grain à moudre aux meilleurs, des fois). Il serait vraiment bien que vous fassiez l'exo 1 à fond (en écrivant le code s'il le faut), et que vous meniez les réfléxions jusqu'à l'exercice 4 en TD (sans écrire le code mais du pseudo-code ?).

En TP, il serait bien qu'ils attaquent le codage de l'exo 4. Pensez à leur mettre un peu de stress la séance suivante pour voir qui a fini le TP. D'ailleurs, demandez leur en début de séance qui a avancé sur le knapsack, et discutez **rapidement** des problèmes rencontrés sur machine avec le TP précédent. Incitez les meilleurs à s'impliquer et à tenter d'aller jusqu'au bout du sujet.

Fin réponse

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, $n_3 = |n_1 - n_2|$

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`.

Réponse

Pas la peine de les laisser bloquer trop longtemps sur ce problème, il n'en vaut pas la peine.

```
// précondition: 1 <= col <= lig <= hauteur
def indiceLigne(lig:Int, col:Int): Int  =  lig * (lig - 1 ) / 2 + col - 1
```

(oui, scala permet de définir une fonction sur une ligne sans accolade de la sorte)

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

Générer les permutations est un code classique qu'il faut avoir vu une fois. C'est une bonne version simplifiée de ce qui vient après. Quelques pistes pour les aider à trouver ce code :

- Énumérer à la main les permutations pour n=3 (321 312 231 213 132 123)
- Faire l'arbre d'appels comme on avait fait pour le knapsack, sauf qu'il n'y a pas un nombre constant d'éléments à chaque point
- Appliquer la recette de récursion classique :
  - Paramètre de récursion : position en cours de remplissage (on a une permutation à sauvegarder)
  - Cas trivial : quand la position est au delà du tableau
  - Aide apportée par le bon génie : remplir correctement les positions suivantes
  - Traitement à l'étage courant de récursion : Pour toutes les valeurs possibles, si une valeur n'est pas encore utilisée, la mettre et remplir le reste.
- (vos idées sont bienvenues pour augmenter la liste)

```
val hauteur = 3 // défini l'instance du problème
val taille = hauteur*(hauteur+1)/2

var permutations:List[Array[Int]] = Nil // Là où on va stocker les permutations

def duplique(src:Array[Int]):Array[Int] = {
  val dst=Array.fill(src.length)(0)
  for (i <- 0 to src.length-1)
    dst(i) = src(i)
  return dst
}

def genere(rang:Int, tab:Array[Int]) { // Génère les permutations
  if (rang>=tab.length) {  // On a déjà tout généré, on arrête
    permutations = duplique(tab)::permutations // On sauve une copie dans la liste générée
    if (permutations.size % 1000 == 0){ // affiche l'état courant pour quand ca dure
        for (i <- 0 to tab.length-1) print(tab(i))
        print(" ")
    }

  } else {                 // Tout n'est pas défini
    for (valeur <- 1 to tab.length) { // pour toutes les valeurs possibles
      var dejaPris = false
      for (i <- 0 to rang-1)
        if (tab(i) == valeur)
           dejaPris = true

      if (!dejaPris) {  // Si elle n'est pas encore prise,
        tab(rang) = valeur  // on la prend
        genere(rang+1, tab) // et on considère les cases suivantes
      }
    }
  }
}

genere(0, Array.fill(taille)(0))

println("Trouvé "+permutations.size+" permutations")
for (tab <- permutations) {
  for (i <- 0 to tab.length-1) print(tab(i))
  print(" ")
}
println
```


CORRECTION

Fin réponse

▷ **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.

Réponse

On peut faire cette fonction en récursif (sur la ligne), mais c'est compliqué pour pas grand chose.

```
def correcte(tab:Array[Int]): Boolean = {
  var permutation = new String()
  for (i <- 0 to tab.length-1) permutation += tab(i)

  for (ligne <- 1 to hauteur-1)
    for (diag <- 1 to ligne) {
      val n1 = tab(  indiceLigne(ligne+1, diag)    )
      val n2 = tab(  indiceLigne(ligne+1, diag+1)  )
      val n3 = tab(  indiceLigne(ligne,   diag)    )

//      println(permutation+"("+ligne+","+diag+"): "+n3+". "+n1+"-"+n2+"="+math.abs(n1-n2))
      if (math.abs(n1-n2) != n3)
        return false
    }
  return true
}
```

Fin réponse

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

Réponse

352164 341265 325461 314562 253614 235416 143625 134526
Notez qu'il n'y en a que 4 d'originales, les autres sont symétriquement égales (dessinez-les).

Fin réponse

▷ **Question 4:** 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**. Que ceux qui n'en sont pas convaincus tentent de calculer la hauteur 5 de cette façon. Moi j'ai craqué avant la fin de la génération de la hauteur 4.

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

Réponse

```
def correcte(tab:Array[Int], rang:Int): Boolean = {
  var permutation = new String()
  for (i <- 0 to tab.length-1) permutation += tab(i)

  for (ligne <- 1 to hauteur-1)
```

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

```
    for (diag <- 1 to ligne) {
      // n2 a forcément l'indice max, pas besoin de tester les autres
      if (indiceLigne(ligne+1, diag+1) <= rang) {
        val n1 = tab(  indiceLigne(ligne+1, diag)    )
        val n2 = tab(  indiceLigne(ligne+1, diag+1)  )
        val n3 = tab(  indiceLigne(ligne,   diag)    )

        if (math.abs(n1-n2) != n3)
          return false
      }
    }
  return true
}
```

Fin réponse

▷ **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 $n_3 = |n_1 - n_2|$

Réponse

```
def genere(rang:Int, tab:Array[Int]) { // Génère les permutations
  if (rang>=tab.length) {  // On a déjà tout généré, solution correcte!

    print("Permutation correcte: ")
    for (i <- 0 to tab.length-1) print(tab(i))
    println("!!")

  } else {                 // Tout n'est pas défini
    for (valeur <- 1 to tab.length) { // pour toutes les valeurs possibles
      var dejaPris = false
      for (i <- 0 to rang-1)
        if (tab(i) == valeur)
          dejaPris = true

      if (!dejaPris) {  // Si elle n'est pas encore prise,
        tab(rang) = valeur  // on la prend
        if (correcte(tab, rang)) // Appel récursif SSI solution partielle correcte
          genere(rang+1, tab) // et on considère les cases suivantes
      }
    }
  }
}
```

Fin réponse

▷ **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 précédemment. 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 éviter d'agrandir des solutions partielles vouées à l'échec. Par exemple, s'il s'avère impossible de placer une valeur dans la case 11 à cause des 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.

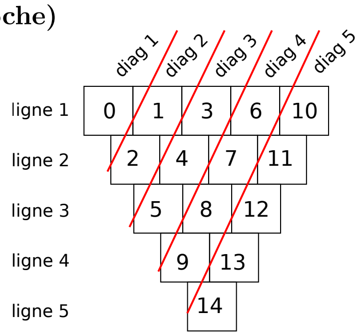


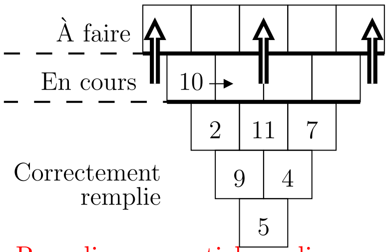
FIGURE 3 – Numérotation diagonale.

L'approche «en colonnes» est préférable car la seconde contrainte lie un nombre aux deux 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.

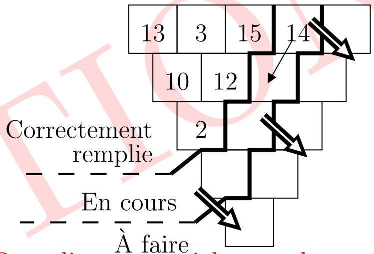
CORRECTION

Réponse

Avant d'aller plus loin, il faut motiver le travail prévu. Il faut faire sentir cette histoire de contrainte au plus tôt pour économiser des générations inutiles. Pour cela, les deux schémas suivants sont utiles :



Remplissage partiel par lignes.



Remplissage partiel par colonnes.

Fin réponse

▷ **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 fonction `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`.

Réponse

Avec le code de indiceLigne , ils devraient parvenir à trouver celle-ci. Mais ce n'est pas la question la plus importante : si le temps manque il est préférable de leur donner pour réfléchir plutôt au reste.

```
// précondition: 1 <= ligne <= diag <= hauteur
def indiceDiag(ligne:Int, diag:Int):Int = diag * (diag - 1 ) / 2 + ligne - 1
```

Fin réponse

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

Réponse

En fait, il suffit de changer l'ordre de parcours et les indices dans le calcul de n_{123}

```
def correcteDiag(tab:Array[Int], rang:Int): Boolean = {
  var permutation = new String()
  for (i <- 0 to tab.length-1) permutation += tab(i)

  for (diag <- 1 to hauteur)
    for (ligne <- 2 to diag) {
      // n3 a forcément l'indice max, pas besoin de tester les autres
      if (indiceDiag(ligne, diag) <= rang) {
        val n1 = tab(  indiceDiag(ligne-1, diag-1)  )
        val n2 = tab(  indiceDiag(ligne-1, diag)    )
        val n3 = tab(  indiceDiag(ligne,   diag)    )

        if (math.abs(n1-n2) != n3)
          return false
      }
    }
  return true
}
```

Fin réponse

▷ **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).

Réponse

Il faut discuter un peu de pourquoi générer() reste la même, mais j'espère qu'ils verront vite qu'une permutation est une permutation, peu importe où sont placées les billes correspondantes dans le triangle.

Voici des benchmarks imprécis réalisés sur ma machine. Ils ne sont pas réalisés avec une rigueur suffisante pour une publication, mais ils sont bien suffisants pour dire que notre nouvelle approche est décevante : on voit pas trop la différence.

hauteur	5	6
par lignes	1.5s	40s
par diagonales	1.3s	39.5s

Fin réponse

★ 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 de 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

Votre code tel qu'écrit à la fin de l'exercice 5 permet de traiter en un temps raisonnable les instances du problème de hauteur 7 ou 8, mais pas vraiment au delà.

▷ **Question 1:** Optimisez votre code autant que possible afin de résoudre l'instance la plus grande possible. Le code le plus efficace connu à ce jour a été proposé par Julien Le Guen, étudiant de la promotion 2008. Il a établi qu'aucune pyramide de hauteur $5 < h \leq 12$ ne respecte toutes les contraintes du problème. Peut-être qu'une solution existe pour des instances plus grandes ?

▷ **Question 2:** Calculez le taux de remplissage maximal pour chaque hauteur de pyramide, c'est-à-dire la quantité de valeurs bien placées dans la solution partielle la plus grande. Pour vérifier que votre code est correct, comparez aux valeurs du tableau ci-dessous. Les temps ont été obtenus sur une machine de 2006 (centrino à 1.5Ghz).

Rang	2	3	4	5	6	7	8	9	10	11	12
Remplissage	$\frac{3}{3}$	$\frac{6}{6}$	$\frac{10}{10}$	$\frac{15}{15}$	$\frac{20}{21}$	$\frac{25}{28}$	$\frac{31}{36}$	$\frac{37}{45}$	$\frac{43}{55}$	$\frac{49}{66}$	$\frac{57}{78}$
Temps	2ms	2ms	3ms	6ms	0,12s	0,9s	6s	1m10s	15m48s	3h12m	1j 5h

Réponse

Ces chronos sont obtenus avec le code en C ultra optimisé écrit par Julien à l'époque (ce code est dans le git du cours). Mais on peut les laisser se décarcasser un peu :)

Fin réponse

Il semble donc que ce problème n'admette pas de solution pour $n > 5$. Pour rendre la recherche plus intéressante, on peut relaxer la première contrainte du problème. Au lieu d'imposer de prendre toutes les valeurs de $\left[1, \frac{h(h+1)}{2}\right]$, on impose seulement de prendre des valeurs distinctes.

Le problème est alors de trouver le remplissage de la pyramide qui minimise l'intervalle dans lequel sont pris les valeurs. Ainsi, la solution recherchée pour $h = 6$ est celle qui utilise toutes les valeurs de [1 ;22] (sauf 15). Pour $h = 7$, seules 3 valeurs sont ignorées tandis que 8 valeurs sont ignorées pour $h = 8$.

▷ **Question 3:** Trouvez les pyramides les plus grandes possibles en respectant ces nouvelles contraintes.

Réponse

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

```
Pyramid of height 6
  6 20 22 3 21 13
   14 2 19 18 8
    12 17 1 10
     5 16 9
      11 7
       4

Ignored elements:  15; Computation time: 0s


Pyramid of height 7
14 31 5 33 32 8 19
 17 26 28 1 24 11
   9 2 27 23 13
    7 25 4 10
     18 21 6
       3 15
        12

Ignored elements: 16 20 22; Computation time: 4s


Pyramid of height 8
  7 33 42 3 44 43 6 29
   26 9 39 41 1 37 23
    17 30 2 40 36 14
     13 28 38 4 22
      15 10 34 18
       5 24 16
        19 8
         11

Ignored elements:  12 20 21 25 27 31 32 35; Computation time: 87s

```

Fin réponse