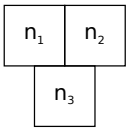
TDP4 : Recursion - Part two (Pyramids)

Module

The objective of this exercise is to continue to practice recursive backtracking on a spicier problem where constrains matter most. The problem is know as the *P*yramid Problem.

Problem statement Lets consider an inverted pyramid (head down) of height h as the one represented below. We want to fill all cells with all integers between 1 and $\frac{h(h+1)}{2}$ so that the following constraints are fulfilled :

1. Each value between $\left[1, \frac{h(h+1)}{2}\right]$ appears exactly once in the pyramid,
2. The value of each cell is equal to the difference of the values in the two cells directly above. Thus, on the figure $n_3 = |n_1 - n_2|$



This problem exists in real situations. The placement of billiard balls is one example (with $h = 5$ in this particular case). Some instances of the problem (some values of h) do not have any solution while others have several solutions (e.g. 8 solutions exist for $h = 3$).

★ **Exercice 1: In-memory representation** An obvious way of representing the pyramid is to use a two dimensional array (`list[list[int]]`). Either all inner lists are the same size (in this case only 50% of the memory will be used) or they are of growing size (in which case we optimize the memory usage somehow.

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

FIGURE 1 – Contiguous memory representation

We will use a one dimensional table (flat-table) by storing the different layers of the pyramid close to each other (one at a time). Given different strategies for cutting these slices, there are numerous ways to number the pyramid cells. In the first approach to the problem, we will use the simplest one, namely following the line numbering of cells as illustrated hereafter.

To access the cells, we need to define a function named `lineIndex(line: int, column:int)->int:` which computes the index of the cell hosting the value at line *line* and column *column*. Note that :
`lineIndex(1,1)=0` ; `lineIndex(2,2)=2` ;
`lineIndex(3,2)=4` ; `lineIndex(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 – Line-based numbering

▷ **Question 1:** Write the core of this function and define all its Pre-conditions
HINT : compute the number of cells in a pyramid of height *line*.

★ **Exercice 2: First approach : a "Generate then test" algorithm**
The first idea (as often when addressing a new problem) is to generate all existing pyramids and once done, check among all candidates, those which satisfy the constraints and those which do not. To this end, one needs to generate all permutations of the list of n first integers, then check those verifying the second constraints [1].

▷ **Question 1:** Design an algorithm that computes the permutations of the first n integers. The associated function shall be named `permutations`.

▷ **Question 2:** Write a function named `correct()` which checks if a given permutation forms a valid pyramid. Start by formalizing the checks to perform and the candidate cells. The `lineIndex()` function will ease your coding.

▷ **Question 3:** Compute all valid pyramids of height 3. If you are not patient enough, you can skip this question and let your code do the job in the next lab-session.

▷ **Question 4:** What is the time complexity of the algorithm doing the correctness test for a solution ?

★ **Exercice 3: Stepwise algorithm design (second approach)**
The solution from the previous exercise is valid but, as you can imagine and later on check in the lab, inefficient. Its inefficiency comes from the fact that it builds all possible solutions even those which do not comply to the constraints of the problem. A possible improvement might be to consider (and check)

1. We go immediately to the second constraint since the first is, in this case, enforced by construction.

the constraints at each step of the solution construction, and when one constraint is broken, immediately stop the exploration.

This is exactly what we saw under the *backtracking algorithms* principles in the lecture last week (surprising, isn't it ? :-).

▷ **Question 1:** To do early checks, we need to adapt a couple of our previous functions. Inspired by the previous `correct` function, write a `correctBis` function that checks only those elements of a configuration that have been placed already (avoiding to check possibles values that have not yet been placed (initialized) in the array to build a solution. To this end, an additional parameter named `rank` is recommended.

▷ **Question 2:** Inspired by the `permutations` function, propose in a function named `validPermutations` an algorithm that stops the build of a permutation as soon as the partially build solution fails to comply to the second constraint : $n_3 = |n_1 - n_2|$.

▷ **Question 3:** During the lab session, compare the execution time of this version versus the previous version of the permutation generation (for various sizes of the pyramid). You can (and are invited to) use the `timeit` library introduced in the previous lab already.

★ Exercice 4: Slice-based generation (algorithm improvement)

Cut the branches leading to invalid solutions turns out to be incredibly much faster than the previous one. It allows to find solutions for pyramids of height 5 in a few seconds. However to address larger problems, we have to refine this approach.

To this end, the objective is to generate the constraints as early as possible to avoid useless generation of invalid partial solutions. For example, if it is impossible to put the value 11 because of constraint violation, cells like 8, 9, 4, 5 or 2 will have been filled for nothing. The objective is therefore to change the order of fillings to detect inconsistencies in early stages and speed solutions search.

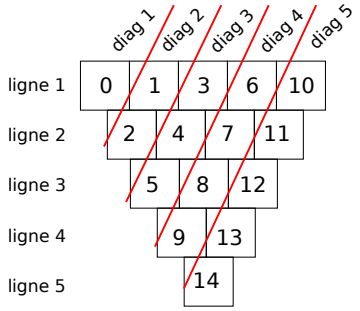


FIGURE 3 – Diagonal numbering.

A column-based approach to the problem appears to be smarter because the second constraint links a value to two others placed above it. It is therefore natural to try to process the value below after the two above it and not the opposite way. This enables, at each step, to check that any valid solution generated from the previous steps will not break the second constraint. As seen in the previous construction, the line-based approach could lead to invalid solutions at layer n where we had to modify earlier layers in the process.

▷ **Question 1:** Rather than building a complicated walk in the pyramid, we simply renumber the cells so that their natural order exposes the constraints as early as possible. Write a `diagIndex()` function close to the `lineIndex()` one but numbering the cells of the pyramid as illustrated in figure 3.
`diagIndex(1,1)=0`; `diagIndex(2,2)=2`; `diagIndex(2,3)=4`; `diagIndex(2,4)=7`.

▷ **Question 2:** Write a new `correct` function named `correcDiag` which checks that the pyramid verifies the second constraint of the problem in the new numbering scheme of the cells.

▷ **Question 3:** In the on-machine lab, compare the execution times of this new version versus the previous one.

★ Exercice 5: Propagation-based generation (further improvement of the algorithm)

When measuring the performance, one can see that the results of the previous improvement are pretty disappointing. So more (and more impacting) improvements are needed. When looking at the way the previously build algorithm works, we can observe that it is totally useless to test all possible values for n_3 and then keep only those which are constraint compliant. In fact, once n_1 and n_2 are known, only one value remains possible for n_3.

Given this observation, one might put a candidate value at the top of the diagonal and simply propagate it which checking that the constraints remain verified (here we should only deal with the first constraint of course).

▷ **Question 1:** Rewrite your algorithm in a recursive way focusing on the diagonal to be filled. At each layer of the recursion, identify which values are candidate and then propagate your choice throughout the diagonal. If the candidate value is already taken elsewhere in the pyramid, it is no longer a candidate :-). We do recommend that you build the following support functions :

— `contains(pyramid:list[int], candidateValue:int, rank:int)->boolean` which returns true if the candidateValue is present in the already filled part of the pyramid (up to cell in the `rank`) ;

— `propagate(pyramid:list[int], candidatValue:int, diagonal:int)->boolean` which tries to propagate the candidateValue along the diagonal through a series of substraction.
— `generateDiag(pyramid:list[int], diagonal:int)` which tries to fill a full pyramid in a recursive way given that all previous diagonals (those which were processed before diagonal are filled and valid. The stop condition of the recursion is that either a diagonal cannot be filled or the entire pyramid is valid.

▷ **Question 2:** On your computer (during the on-machine session), verify that this version finds all the solutions found by the previously designed algorithms. Check for heights 5,6, and 7. Be aware that no solution exists for height 6 nor for height 7.

★ **Exercice 6: One step beyond ! (the Madness bonus track)**

Your code, as written in the previous exercise (with all its early constraints checking) works fast enough to address in a reasonable delay, problem instances of up to height 7 or 8 (even 9 of faster machines), but not beyond (hence the title :-).

▷ **Question 1:** Optimize your code as much as possible to solve the biggest possible instance. Our most efficient code known today was proposed by Julien Le Guen, a 2008 graduate (yes some 15 years ago) He showed that there is no solution for any pyramid having a height $5 < h \leq 12$. Maybe this is not true for bigger pyramids ?

▷ **Question 2:** Compute the maximal filling rate for each pyramid height (the number of well placed values in the best partial solution). To check the correctness of your code, you can compare the results to the following table. You might reach faster times since those were obtained on a machine back in 2006 with a 1.5Ghz Centrino processor.

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

It looks like this problem does not allow any solution for $n > 5$. To make the exploration more interesting, we need to relax the constraints. This might enable the problem to have solutions even for larger sizes. For each question below, find the tallest possible pyramids respecting those news constraints. If you find better ones that the current state of the art given below, please share them with us. We will guarantee that you will become famous in CS54, and beyond (recursively) !

▷ **Question 3:** It is always frustrating to explore solutionless problems. To allow the emergence of solutions, a first idea is to relax the first constraint. Rather to impose that all values in the pyramid are bounded by $\left[1, \frac{h(h+1)}{2}\right]$, lets state that we only require that values in the cells are distinct. The problem is now to find a filled pyramid which minimises the interval in which the values it holds are taken. Thus, for a pyramid of height $h = 6$, the solution is the one using all values in $[1; 22]$, except 15. Back in 2006, Julien Le Guen also studied this version of the problem and found solutions for $h \in 6, 7, 8$ which ignore respectively $1, 3$ and 8 values.

▷ **Question 4:** Another variant of the problem is to no longer put an absolute value in the constraint computing and allow negative numbers to be inserted in the cells. This variant has never been studied (to our knowledge). So if you wan to see your name appear in future lab exercises, have fun !