The Education Column

by

Juraj Hromkovič

Department of Computer Science
ETH Zürich
Universitätstrasse 6, 8092 Zürich, Switzerland
juraj.hromkovic@inf.ethz.ch

Teaching Recursion and Dynamic Programming Before College

Michal Anderle
Comenius University
anderle@dcs.fmph.uniba.sk

Michal Forišek
Comenius University
forisek@dcs.fmph.uniba.sk

Monika Steinová
monika.steinova@alumni.ethz.ch

Abstract

This paper is about teaching the algorithmic concepts "recursion" and "dynamic programming" earlier than in a college undergraduate Algorithms 101 course. We describe our motivation to do so, we provide background on how we believe it should be done and why, and we discuss our practical experience with doing so.

1 Overview

Within the context of this paper, *recursion* denotes the use of self-reference, most notably in mathematical definitions and in algorithm design. Two useful canonical examples are the definition of an ancestor (your ancestors are your parents and their ancestors) and the MergeSort algorithm.

The term *dynamic programming* refers to all kinds of computation where solutions to multiple instances of the same problem are computed, stored, and later reused. Problems solvable via dynamic programming are said to exhibit an optimal substructure – i.e., optimal solutions to larger instances of the problem are related to optimal solutions to other, smaller instances of the same problem.

1.1 Teaching Recursion

Teaching recursion is already a well-researched topic. Rinderknecht [15] extensively surveys the research on teaching and learning recursive programming over a period of more than 40 years. Another review of this area which focuses on publications that document research results is given by McCauley et al. [14].

1.2 Teaching Dynamic Programming

Dynamic programming is a standard paradigm used in the design of efficient algorithms. This approach is usable for problems that exhibit an optimal substructure: the optimal solution to a given instance can be recursively expressed in terms of optimal solutions for some sub-instances of the given instance.

There are two main ways to implement algorithms that use dynamic programming. These two ways are essentially isomorphic, but there are subtle differences that may make one approach more suitable than the other, depending on the exact setting. The two approaches are *top-down, recursive* (usually called *memoization*) and *bottom-up, iterative*.

Teaching dynamic programming sooner than in college has so far received very little attention, and it is mostly considered a topic that is too difficult for undergraduate college students.

The paper by Forišek [7] surveys the way dynamic programming is explained in standard college textbooks [4, 16, 5, 8, 17], identifies and analyzes multiple issues with those expositions, and presents a detailed suggestion on how dynamic programming should be taught. The most notable feature of the presented approach is that it no longer treats the design of an algorithm using dynamic programming as a conceptual black box – instead, our approach intentionally breaks up the design of a dynamic programming algorithm into multiple smaller and easier conceptual steps. The approach presented in this paper has been used with much success when teaching dynamic programming in an introductory university course, as well as when teaching it to talented kids aged 15–19.

Böckenhauer et al. [3] describe their experience with teaching dynamic programming in a way that can be introduced to first-semester students of natural sciences with almost no background in computer science. In order to do this, they focus solely on iterative dynamic programming with a two-dimensional matrix.

Erdősné Németh and Zsakó [6] describe a set of tasks that can help kids build various problem-solving strategies, and in particular the dynamic programming concept. Their tasks range from activities in the style of CS Unplugged, through Bebras tasks, all the way to secondary-school algorithmic competitions.

1.3 The Algorithmic Competition PRASK

Slovakia has a rich history of various algorithmic competitions. They differ in the types of tasks, ages of participants, and the length of the competition. A new addition to them is the competition PRASK, founded in 2015. This competition is focused on talented middle schoolers (approximate ages 11–15) with interest in computer science. The main objective of PRASK is to promote and develop algorithmic and programming skills. The competition also serves as a gateway towards high school informatics competitions such as the Slovak National Olympiad in Informatics. The tasks in PRASK are designed to be approachable by beginners without previous knowledge in computer science.

PRASK is a long-term competition that consists of multiple rounds, spread across one school year. In each round the contestants have approximately one month to come up with their solution to a given set of five problems. There are various types of tasks: theoretical (the solution is a written text), practical (using the tools available on a regular computer), or programming (the solution is a program in an existing language). All problems presented in this paper have been at some point used as *theoretical* tasks in this competition.

A more detailed report on the competition is given by Anderle [1].

2 Our Approach and Goals

We believe that the concepts of recursion and dynamic programming are important concepts in algorithm theory. Both have often been documented as hard to approach and unintuitive, even for university students. We are convinced that the main reason for this is that those students didn't have sufficient prior exposure to problems and settings that introduce many of the necessary concepts in a natural and intuitive way.

In this paper we present a carefully selected collection of problems that can introduce recursive thinking and concepts related to dynamic programming. All problems shown in this paper were successfully solved by talented kids aged approximately 12–15, and are pretty approachable for the general population of kids aged 15–19.

Note that (as opposed to, e.g., Forišek [7]) our goal in this paper is *not* a complete exposition of dynamic programming as an algorithm design technique. The problems presented in this paper provide the solver with individual schemata related to the technique. At a later age, when learning the technique as a whole, the students will then easily discover the links between these schemata, and they will be able to infer general patterns and acquire a deeper understanding of the topic.

3 Problems

In this section we present the collection of problems we selected as suitable for introduction of various concepts related to recursion and dynamic programming. Each subsection contains a brief overview of an algorithmic topic followed by one or two sample problems. Each problem consists of a simplified task statement, a discussion of its algorithmic background, and selected experiences from using the problem as a task in the PRASK competition.

Note that the task statements used in the competition are much more verbose – e.g., they include illustrated examples where applicable and they usually include a short story that provides additional motivation to solve the task. In this paper these parts were omitted to conserve space. For easier orientation in the text, problem statements are typeset with a vertical rule on the left.

3.1 Game of NIM Variants

The name NIM [2] denotes a class of impartial combinatorial games for two players that are played by removing tokens according to given rules. A player loses the game if they cannot make a valid move.

We view these games as an important didactic tool. One of the main reasons is that kids like to play games, and actually playing small instances of these games against each other is a great way to discover how they work and build intuition that can later be generalized to a full solution.

The main reason for including these games in this paper is the inherently recursive nature of their solution: In an impartial combinatorial game each position is either *winning* (i.e., the player to move has a winning strategy) or *losing*. The evaluation of a position can be phrased using two simple recursive rules:

- If each move from a position P leads to a winning position, position P is losing. (Note that this includes the base case for the recursion: the positions where you lose immediately as you have no valid moves left.)
- If there is a move from position P that leads to a losing position, position P is winning (and that move is a winning move).

Sample Problem Statement 1

> **The Game with Sparklers.**
> Petra and Janko have a box of sparklers. They use it to play a game. The two players take alternating turns in which they remove some sparklers from the box (and light them). Exact rules for removing sparklers are different in

different subtasks. Petra goes first. The player who is forced to remove the last sparkler loses the game (and has to go throw out all the burned-out sparklers).

Subtask A: There are 7 sparklers in the box. The kids can remove 1, 2, 3, or 4 sparklers in each turn. How should Petra play to win the game?

Subtask B: The same rules, but there are 42 sparklers in the box. Also for this game Petra has a winning strategy – i.e., there is some set of rules how she should play such that if she follows them, she is guaranteed to win the game regardless of how Janko is playing. Find and describe a winning strategy for Petra. Justify why your strategy works.

(An example of a strategy would be "if Janko just took 3 sparklers, you take 1, otherwise you always take as many as you can". However, this particular strategy is not a winning strategy, Janko can beat it.)

Subtask C: Petra and Janko are now going to play 20 independent games, one after another. Game x will be played using a box that contains x sparklers. In these games, the allowed moves are to remove 1, 3, 4, or 5 sparklers (but not 2). For each game determine who wins if both kids play optimally.

Subtask D: The same game as in Subtask C, but with a box of 2015 sparklers.

Analysis

The above problem is a *misère* version of a game of NIM. (To map it to the regular version, one can simply label the position with one sparkler left as losing.) The statement is designed to gradually introduce the concepts needed for the analysis of the game. The strategy for subtask A is easy to discover by playing out all possibilities, and its generalization for subtask B is really easy to describe ("start by taking 1 sparkler, then whenever Janko takes x you take $5 - x$").

In particular, note that the way subtask C is phrased motivates the solver to discover how to *reuse* information – "This move would get me to position x, and I already know that the player to move wins from that position." – which is precisely the application of dynamic programming.

Experience

A slightly modified version of the above problem (with a more verbose statement that explained in more detail the concept of playing optimally) was used in a round of the PRASK competition [9].

We received 18 solutions. Almost all contestants successfully solved subtask B, with five of them also explaining in detail how they derived it by starting from the smallest positions and noticing that 1 and 6 were losing.

In subtask C the full statement explicitly explained what happens in the trivial game with one sparkler to avoid confusion. Twelve contestants solved this

task, usually by writing out a table, noting which positions are losing and saying that from other positions the player should move to those. Other contestants either didn't send anything, or they worked out small instances but then made an incorrect generalization.

The last subtask was the hardest, only solved by eight contestants. Most of the others also noticed the pattern but failed to generalize it correctly. Even those who solved it correctly struggled to justify why the observed pattern has to repeat.

It seems that the task was interesting and accessible for the contestants. Most of them were able to solve the main parts of it and come up with dynamic programming necessary for solving it efficiently. In the statement as presented above we made two improvements based on the experience with the task used in the contest: we added the introductory subtask A, and we changed subtask D to use the same set of moves as C.

Sample Problem Statement 2

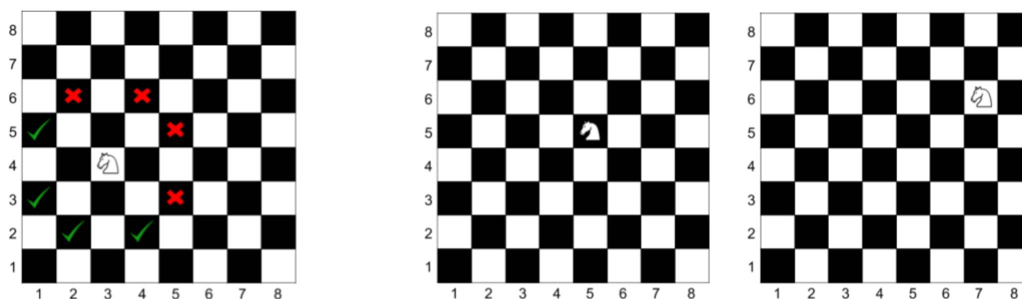

Figure 1: Valid moves of a knight (left). Subtask A (middle). Subtask B (right).

> **Knight on a Chessboard**
>
> Kika and Andrej are playing a game using a regular chessboard and one knight. The knight is placed on some cell of the chessboard, then players take alternating turns. In each turn, a player has to move the knight according to chess rules to a cell that is closer (in terms of Manhattan distance) to the bottom-left corner (see Figure 1 left). The player who cannot move the knight loses.
>
> **Subtask A:** The knight starts at $(5, 5)$ (see Figure 1 middle). Who can win the game, no matter what their opponent does? Provide a description of how that player can win.
>
> **Subtask B:** Answer the same question if the knight starts at $(6, 7)$ (see Figure 1 right).

Subtask C: For each cell of the (8×8)-chessboard determine whether Kika (the starting player) can guarantee a win if the knight starts at this cell. Fill a two-dimensional table. Into the cell (r, c) write 1 if Kika has a winning strategy if the game starts with a knight on the cell (r, c), or 0 if she does not. Describe how you fill the table.

Subtask D: Generalize the algorithm to an arbitrary $(r \times c)$-chessboard.

Subtask E: A harder version of the game uses multiple knights. Each cell of the chessboard may contain arbitrarily many of them. In each turn, the active player can choose an arbitrary number of knights (but at least one) and move each of them according to the previous rules.

Explain how to evaluate a position in this game. That is, given the coordinates of all knights, how do you determine who has a winning strategy? Can the table computed in Subtask C help us?

Analysis

This is also an impartial combinatorial game. It's not exactly a game of NIM, but they share a common generalization (games that move a token on an acyclic graph) and they share the same properties that make the game a useful didactic tool. Additionally, this game is based on a familiar setting and has an easily describable state. The problem given above is designed gradually, moving from specific examples to more general solutions.

Starting positions in subtasks A and B are carefully chosen. The one in subtask A is losing, and it is expected that most solvers will determine this by an exhaustive search. The one in subtask B is winning, and there is a simple argument why: the move to $(5, 5)$ which we know to be losing. This connection can lead pupils towards reusing known information about the previous states.

Subtask C requires a thorough approach while filling the table. The statement strictly determines the format of the output. This is intentional because, at this stage, solvers need to move from thinking about specific players to thinking about the *player to move*. Number 0 and 1 represent this idea better than the characters 'K' and 'A' that could be chosen by the contestants. Subtask D then leads to a better formalization of the discovered relations.

The last subtask is optional, intended as possible follow-up. It requires a better understanding of winning and losing positions and shows how complex problems can be solved by looking at easier instances. (For the full solution, consider what happens if all the knights are on positions that were losing in the one-knight version of the game, and what happens in all other cases.)

Experience

This exact task, of course with a more verbose statement, was used in a round of the PRASK competition [11]. We received 12 solutions to it.

The first two subtasks were successfully solved by all contestants. All but one then generalized their observations and described the rules for winning and losing positions. Most of them also described how they essentially used dynamic programming to gradually fill the resulting table and how they would generalize it for bigger chessboards. But instead of doing it cell by cell, they usually filled the table in "waves" of losing and winning positions.

One contestant, even though she described the rules correctly, failed to produce a correct result. It seemed that she filled some cells before she filled all reachable positions. The problem first occurred at position $(5, 3)$, which is why we would consider adding it to the early subtasks when reusing the problem.

The last subtask was solved by 8 contestants, others didn't write anything to their solutions. All of them correctly generalized the problem and produced a correct strategy.

3.2 Coin Change Problem Variants

Using money to pay a sum exactly is an excellent environment, because already at a young age contestants have practical experience with it and thus they can benefit from the intuition they already have, and they don't have to use abstraction when reasoning about problems set in this environment.

This environment gives us a class of problems related to the famous "knapsack" problem. In these problems we ask questions like "given a sum and a set of denominations, in how many different ways can we pay the sum?" or "given a sum and a set of denominations, what is the smallest number of coins and banknotes needed to pay the given sum exactly?" All of these problems are solvable in pseudopolynomial time using a natural application of dynamic programming.

The optimal substructure comes from the following observation: When paying any amount x, I have to use some coin $y \leq x$ and then I'm left with the remainder (which is $x - y$) and the same question as before.

Sample Problem Statement

> **Money Problems.**
> In Absurdistan they use gold coins as the only form of payment. The coins come in multiple denominations.
> When paying any amount, people of Absurdistan like to use a simple greedy algorithm: always take the largest coin that does not exceed the sum you are

trying to pay.

A set of coin denominations is *good* if it has two properties:

1. If one has an unlimited number of coins of every denomination, one can pay any positive integer amount exactly.
2. Using the greedy strategy to pay any amount will always use the fewest coins possible. (That is, there cannot be a different way to pay the same amount using fewer coins.)

For example, the Euro denominations and the US dollar denominations are two examples of good sets of coins.

Subtask A: Design a set of coin values (each between 1 and 7, inclusive) that will *violate both* properties. Explain why that's the case.

Subtask B: Absurdistan currently uses coins with values 1, 4, 7, and 19. This set of coins is *good*, but people in Absurdistan still struggle to use the coins. Help them by creating a simple table: for each amount from 1 to 20, the table should contain the smallest number of coins with which it is possible to pay that amount exactly. (Optionally, you can also include one optimal way of paying each amount.)

Subtask C: Create a similar table for the set of values you designed in Subtask A. (Of course, in this table at least one amount will be marked as impossible to pay exactly.)

Subtask D: Absurdistan will eventually have a money reform that will introduce a new set of denominations. Describe a general algorithm they should use to create a new table such as the one you created in Subtask C. Try to find an algorithm that is fast enough so that we can use it to create a table for amounts 1 to 500 if the number of new denominations is at most five.

Subtask E: After the money reform the king realized that coins with value x have his face on them, but coins with value y bear the face of the previous king (whom the current king hates). Therefore, the current king made a royal decree: whenever anyone is paying any amount, they must always use at least as many x-coins as y-coins.

Describe an algorithm that can be used to compute our table (i.e., for each amount, how many coins are needed to pay it) if the royal decree is in effect.

Analysis

The first subtask is designed to help the solver familiarize themselves with the greedy strategy and to discover why it's not necessarily optimal. It's pretty obvious that we need a coin worth 1 to pay the amount 1, and that once we have such a coin we can pay any amount. A simple set of coin values for which the greedy algorithm fails is the set $\{1, 5, 7\}$: e.g., the greedy algorithm will pay the sum 10 as 7 + 1 + 1 + 1 (four coins used) whereas the optimal way is 5 + 5.

In Subtask B the contestants can practice using the greedy strategy to pay various amounts. Already this subtask offers them an option to discover a very simple form of dynamic programming: instead of going through the whole algorithm for each sum, we can reuse the values computed sooner. For instance, when paying the sum 29 we will start by using the coin 19, so then we are left with the sum 10 *and we already know how to pay that sum optimally*.

Subtask C is still small enough to be solved by exhaustive search, but the benefits of using dynamic programming to speed up your work are much more significant when the number of ways to pay a sum grows exponentially.

Subtask E is solvable using a clever trick: we can guarantee satisfying the royal decree by simply gluing one x coin to each y coin. Thus, we get the same problem as before, only the set of denominations now contains $x + y$ instead of y.

Experience

An essentially identical task was used in the PRASK competition [13], the only differences were that in Subtask A the upper bound was not present and in Subtask B it was not highlighted that the given set of denominations is good.

All contestants who attempted this problem solved Subtask A successfully, with one of them remarking: “Let’s come up with a set where the second property fails, then we just multiply all its coins by 2 to make it also fail the first property.”

Among contestants who attempted Subtask C, one half still used some brute force approach, the other half managed to discover some version of dynamic programming. (When reusing the task we would raise the maximum amount to pay from 20 to 50 in order to discourage the brute force approach.)

Notably, two different approaches (corresponding to different orders of `for`-cycles in the implementation) were both present: some contestants added the coin values one at a time, each time recomputing the optimal solution for all amounts, others were iterating over all amounts and always attempted to use each coin as the first one when paying the current amount.

The best two contestants (ages 14 and 15) managed to fully solve the task, including the trick needed to solve Subtask E.

We note that the first of the two approaches mentioned for Subtask C can be extended to a new subtask: given a finished table for some unknown set of denominations and a new coin type, how will the table change?

3.3 Monotonous Paths on a Grid

One of the classes of problems that are most suitable as early introductions to the concept of dynamic programming are problems that involve paths on a grid, with Manhattan being a common metaphor. The canonical problem from this category

is: “You are going from an intersection to another intersection that is r blocks to the south and c blocks to the east. Assuming you can only walk southwards and eastwards, how many different routes can you take?” Solving this problem first in its pure form and later in the presence of obstacles that block specific roads and/or intersections is a really easy problem and again, the application of dynamic programming is very natural: “all routes that reach the goal = all routes that reach it from the north + all routes that reach it from the west”.

Below we present one of our problems that falls into this broad category.

Sample Problem Statement

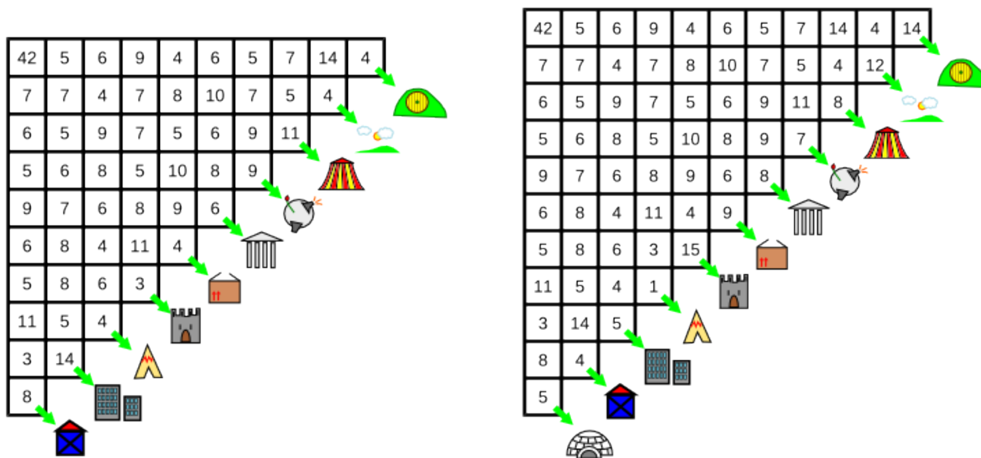

Figure 2: The candy-giving stations and the dwellings of ten travellers (left). The same situation extended by one more diagonal and one more dwelling (right).

Collecting Candy.

We have a “triangular” grid of cells. (See Figure 2 for two examples.) A group of travelers is located in the top-left corner. Each traveler is on their way home. The homes are depicted below the diagonal. In each step, each traveler can move either one cell down or one cell to the right. Each cell contains a station that hands out a fixed amount of candy to each passing traveler.

Subtask A: Consider the plan shown in Figure 2 on the left. For each dwelling, determine the *maximum* total amount of candy that can be gathered on the way from the top-left corner to that dwelling. Describe how you computed your answers and explain why they are correct.

Subtask B: The plan shown in the same figure on the right is the same, except for an additional diagonal of cells with stations. Again, determine the optimal candy count for each dwelling. Can we reuse the solution of Subtask A to answer this question easily? If yes, how?

Subtask C: Knowing the optimal amounts is not enough! For each dwelling from Subtask B, find one optimal path along which the traveler should go in order to collect as much candy as possible. In detail, describe the process of how you determined these paths.

Subtask D: Suppose we gave you a similar grid with 80 rows. Would you still be able to solve the same task as on the grid with 10 rows? Try to estimate the amount of time you would need to solve such a big grid.

Analysis

The straightforward visualisation on a grid makes the task approachable also for our young students. The candy counts written in cells lead students into thinking about writing other numbers into cells when computing their solutions.

We have broken the task into several steps through which we intended to gradually build students' experience and abstraction, enabling them to solve the last, most general task.

Subtask A contains a specific instance of the problem to increase the students familiarity with the task. Since the students were given weeks to find a solution, we intentionally chose a fairly large grid by which we aimed to discourage students from trying all potential paths manually. (This is still possible, e.g., for the second dwellings from the top and bottom to verify the results obtained by a different method.)

Subtask B is actually giving useful advice on how to approach Subtask A as well – by guiding the students towards processing the instance one diagonal at a time.

Subtask D should encourage the students to generalize the process they used manually to solve the specific instances. At this stage (if not earlier) we expect the students to claim that each cell is reachable from at most two other cells (from above and from the left) and we can always determine the maximum by picking the better of those two options.

This subtask also guides them towards a future concept of (asymptotic) time complexity of algorithms, and we expected an argument that the computation time is proportional to the number of cells, and that the new grid has roughly 60 times more cells than the old one.

3.3.1 Experience

This exact problem was used in a round of the PRASK competition [10].

One contestant only submitted numeric answers without any explanation. All others employed a visualisation of the computed results in a grid. The majority stored candy counts in the cells, but we had a single student that moved along grid edges of his figure and stored the counts in the crossroads. Both directions of computation (both “which cells influence my current cell” and “which cells are influenced by my current cell”) were present among the solutions.

A range of different approaches could be observed in Subtask A, including correct solutions but also:

- An incorrect heuristic that from the view of consecutive 2–3 stations picks the one which yields the most candy and moves on to another 2–3 stations.
- A greedy approach which processes the dwellings gradually and prefers cells with a maximal candy count in a row or a column.
- An application of Dijkstra’s algorithm[1].

In general we were able to observe that students were building their knowledge gradually. We saw few students that analyzed grids consisting of an increasing number of rows (and all columns). Several students were able to generalize these ideas further but the majority were thinking in terms of the fixed instance given in the subtask.

The transition from Subtask A to Subtask B saw mixed results. On the one hand, all the students were able to reuse their solution of Subtask A to determine the counts for the bigger instance. On the other hand, some students did not solve Subtask A correctly and did not fix their solution retroactively using their new observation. We conclude that these students have not fully realized that in a similar spirit smaller instances could have been used in their calculations in Subtask A.

Only the students who solved Subtask B correctly were also able to provide a correct solution to Subtask C. Additionally, they all computed the path together with the candy count calculation.

We correctly predicted that the generalization is a difficult leap. Some students able to solve the given instances using previous cells were unable to detach from the fixed size and formulate the idea for an instance of arbitrary dimension. Only the best two contestants were able to estimate the time complexity correctly.

If the task were to be used in a class, we recommend shrinking the instance used in Subtask A to allow brute force solutions, and also shrinking Subtask B to match.

[1] The student claims that the algorithm was mentioned in one of his textbooks and he studied it more carefully to solve the subtask.

3.4 Fractal Patterns

The visual nature of fractals makes them a popular tool when familiarizing young students with recursion. Below we describe a slightly more involved task that references one particular fractal pattern.

Sample Problem Statement

> **Pyramid with Sarcophagi**
>
> Egyptians decided to build a new Great Pyramid and move all sarcophagi into it. The pyramid will have a triangular cross-section. Floors will be numbered from 1 on the top and rooms on floor x will be numbered from 1 to x from the left to the right. A local custom dictates that only some rooms are suitable for a sarcophagus. The rules for this are as follows:
>
> The room on the top must contain a sarcophagus.
>
> Each other room must contain a sarcophagus if and only if there is *exactly* one sarcophagus in the one or two rooms that are directly above this room.
>
> These rules uniquely determine which rooms must and which rooms must not contain a sarcophagus. The first six floors of the pyramid are shown in Figure 3.

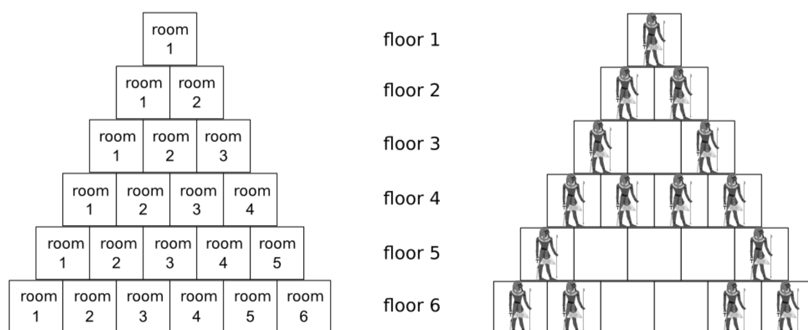

Figure 3: The locations of sarcophagi in a 6-floor pyramid.

> **Subtask A:** Notice that each room on the 4th floor contains a sarcophagus. Imagine that we are constructing more and more floors of the pyramid. Will there be other floors that are fully filled? What numbers will they have and why? Is there an infinite number of such floors?
>
> **Subtask B:** For each of the following rooms, determine whether they contain a sarcophagus: room 10 on floor 20; room 266 on floor 276; room 11 on floor 276.

Subtask C: Devise a procedure that will compute whether a given room on a given floor contains a sarcophagus. Try to find a way that will be as fast as possible. There is a solution that can determine the answer for any room in a 1 000 000-floor pyramid in a minute using just a pen and some paper.

Use your procedure to find out whether room 498 on the 2019th floor contains a sarcophagus.

Analysis

The task is based on a well-known structure – the Sierpinski triangle. This discrete version of the triangle is recursive and has a nice visual interpretation, which makes it accessible to the pupils. We define it by using the recursive definition of Pascal's triangle (i.e., binomial coefficients) and taking just the parity of these numbers.

The first subtask is a bit unconventional, as it asks for the general proof right away. The reasoning is to give pupils the opportunity to explore the recursive structure of the Sierpinski triangle. The resulting pattern (powers of 2) is easy to spot when the picture is extended by more rows and it is closely related to the overall structure. The proof is also fairly simple to formulate.

Afterwards, pupils are asked to solve a few specific instances in subtask B. The first instance is small and solvable by writing out 20 rows of the triangle. The second is larger and requires the use of previously obtained conclusions. Also, the position $(266, 276)$ is the same as $(10, 20)$ just in another part of the triangle. The position $(11, 276)$ is the same as $(266, 276)$ due to the vertical symmetry of the pyramid.

The last subtask asks pupils to generalize the process used in the previous subtask. However, some notion of effectivity needs to be introduced to avoid solutions that will just construct the whole triangle. And because pupils don't know time complexity, intuitive concepts such as "using pen and paper" were incorporated into the statement.

Experience

Almost identical tasks, with a more detailed statement and some small modifications, were used in one round of the PRASK competition [12]. In this round, 15 contestants sent a solution to it.

With a single exception, all contestants were able to find the pattern of fully filled floors, but only 6 of them proceeded further and correctly proved it. However, from the analysis of the solutions it was obvious that at least 6 more contestants correctly understood the recursive structure of the Sierpinski triangle. It is

hard to tell whether they didn't know how to formulate the correct proof or didn't find it necessary to do so.

Subtasks B was solved by 12 contestants and there weren't any obvious problems. The only mistakes stemmed from a wrong generalization of the pattern, but mostly, pupils were able to use the structure of the triangle and figure out correct answers for both positions.

Eleven contestants attempted to solve the last subtask. Two of them used a wrong pattern, one just described how the structure of the triangle looks but didn't provide any procedure. Several of them were aware of the connection to binomial coefficients (as far as we can tell, they knew this connection beforehand, but weren't aware of the recursive structure) and approached this subtask by trying to evaluate the parity of the corresponding binomial coefficient. Around half of the contestants ended with a correct and efficient solution.

The simple recursive structure of the discrete Sierpinski triangle makes it suitable to be explored by middle school pupils. The efficient algorithm to determine the value of a given cell is reasonably simple, but its effectivity is hard to analyze for pupils of this age. Hence, we did not ask for such an analysis and instead we just provided a large instance for them to solve manually in order to gain some intuition about the algorithm's complexity.

4 Conclusion and Acknowledgement

In this paper we have presented examples of tasks for pre-college students that motivate recursive thinking and dynamic programming techniques. The tasks consist of several subtasks with gradual difficulty ranging from concrete instances to abstract tasks. The students are lead through these subtasks and build more abstract conclusions and discover the recursive nature of the problems. We do not aim at completely covering the techniques but we rather deepen the understanding of how to reuse smaller solutions on the way to solve large instances.

We are thankful to the whole collective of volunteers who help run the Slovak programming competitions and other extra-curricular activities for talented kids in STEM areas for all their contributions. In particular, we are thankful to people who helped with the problems mentioned in this paper when they were used in the PRASK competition: Marián Horňák, Ján Hozza, Kristína Korecová, Andrej Korman, Mário Lipovský, Roman Sobkuliak, and Mária Vajdová.

References

[1] M. Anderle. PRASK – An Algorithmic Competition for Middle Schoolers in Slovakia. *Olympiads in Informatics*, 12:147–157, 2018.

[2] E. R. Berlekamp, J. H. Conway, and R. K. Guy. *Winning Ways for your Mathematical Plays*, volume 1. Wellesley, Massachusetts; A. K. Peters Ltd., 2 edition, 2001.

[3] H.-J. Böckenhauer, T. Kohn, D. Komm, and G. Serafini. An Elementary Approach Towards Teaching Dynamic Programming. 128:587, 06 2019.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.

[5] S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani. *Algorithms*. McGraw-Hill, 2006.

[6] A. N. Erdösné and L. Zsakó. The Place of the Dynamic Programming Concept in the Progression of Contestants' Thinking. *Olympiads in Informatics*, 10:61–72, 07 2016.

[7] M. Forišek. Towards a Better Way to Teach Dynamic Programming. *Olympiads in Informatics*, 9:45–55, 07 2015.

[8] J. Kleinberg and É. Tardos. *Algorithm Design*. Addison-Wesley, 2006.

[9] KSP. Prskavkový problém. https://prask.ksp.sk/ulohy/zadania/996, 2015. Accessed: September 2019.

[10] KSP. Párty v časopriestore. https://prask.ksp.sk/ulohy/zadania/1022/, 2015. Accessed: September 2019.

[11] KSP. Prehra na šachovnici. https://prask.ksp.sk/ulohy/zadania/1354/, 2017. Accessed: September 2019.

[12] KSP. Premiestnení faraóni. https://prask.ksp.sk/ulohy/zadania/1354/, 2017. Accessed: September 2019.

[13] KSP. Potiaže s peniazmi. https://prask.ksp.sk/ulohy/zadania/1741, 2019. Accessed: September 2019.

[14] R. McCauley, S. Grissom, S. Fitzgerald, and L. Murphy. Teaching and learning recursive programming: a review of the research literature. *Computer Science Education*, 25:37–66, 01 2015.

[15] C. Rinderknecht. A Survey on the Teaching and Learning of Recursive Programming. *Informatics in Education*, 13:87–119, 04 2014.

[16] R. Sedgewick. *Algorithms in C++*. Addison-Wesley, 3rd edition, 1998.

[17] S. S. Skiena. *The Algorithm Design Manual*. Springer-Verlag, 2nd edition, 2008.