A Problem-based Curriculum for Algorithmic Programming

NIKHÁZY László

Abstract. Engagement of students plays a crucial part in education, even if they are gifted children. We know a success story: the extracurricular mathematics camps of Lajos Pósa for talented teenagers in Hungary. The key to that success is the excellently engineered network of problems that guide students through discovering the world of higher-level mathematics. It would be a novel approach to teach computer programming and algorithms similarly. In this paper, we attempt to design a network of problems selected specifically for discovery learning of algorithms and data structures from beginner to advanced level, targeted for secondary and high school talented students. This could serve as the curriculum for extra classes or camps conducted with the problem-based teaching method we describe.

Keywords. talent education, competitive programming, discovery learning, algorithms and data structures.

1. Introduction

There is a unique system for mathematics talent education in Hungary, led by mathematician Lajos Pósa and his students. The core element of this system is the series of camps in which gifted pupils can explore mathematics with inquiry-based learning [1]. It is the author's goal to establish a similar initiative in the field of computer science. Within computer science, we focus on the core programming skills by teaching algorithmic programming from the beginner to the highest level.

Algorithmic programming involves dealing with well-defined problems to which the solution is an algorithm that calculates the desired output from the given input, and the program is a way of expressing this algorithm that allows executing and verifying the solution on a computer. Correctness and effectiveness are both key measures that are evaluated by extensive testing of the programs. There are a lot of excellent resources available online which promote learning algorithmic programming on an advanced level, for example Halim's book [2], and the massive problem base of past Codeforces contests [3]. However, to use them for our educational goals, we need to organize these materials and exercises in such a way that enables learning through a series of problem-solving.

In this discovery learning scenario, we would like to create situations in which students are facing a problem, and they have already seen the key ideas leading to the desired algorithm while solving different tasks previously. Therefore, the main challenge of the teacher is designing the curriculum and arranging the exercises in a proper structure that makes it possible to introduce the right problem at the right time. In this paper, we present a system of topics and methods, accompanied by exercises, that could serve as the curriculum for extra classes or camps conducted with the problem-based teaching method that we already use in individual and small group programming lessons for high school students.

The structure of this paper is organized as follows. In chapter 1, we briefly introduce the mathematics camps and the didactics of Pósa in the context of discovery learning. In chapter 2, we describe the goals and challenges of adapting the Pósa-method for computer programming talent education and elaborate on the design and of the curriculum. In chapter 3, we present the curriculum in detail, list of topics grouped to units and tables with the collection of tasks for each of them. In chapter 4, we show an example of a topic, dynamic programming followed throughout the entire curriculum. Chapter 5 contains a short summary of the presented content.

1.1. Discovery learning

The term discovery learning refers to pedagogical methods, in which students learn through their exploration of a certain topic. The goals are usually threefold:

- acquire deep knowledge,
- develop cognitive skills,
- increase engagement.

Deep knowledge in this sense means that the learned information, concepts and methods have very strong roots in the long-term memory, thus the person has a higher level of understanding of the subject and can apply this knowledge more successfully in new situations. During the learning process, students spend most of the time actively working individually or in groups, their knowledge is constructed by themselves through these activities. This requires immerse brain capacities used in a variety of forms, through which cognitive skills develop highly. Students' joy is a key value in the whole process, to motivate them for further participation and increase their endurance.

Discovery learning is strongly related to the constructivist learning theory, which relies on the assumption that people construct their knowledge during mental activities. The learners are considered organisms that seek meaning, and reflecting on their experience, derive their own set of rules and mental models of the world. Numerous educators apply discovery learning in modern education. Wouter van Joolingen [4] describes it as "*a type of learning where learners construct their own knowledge by experimenting with a domain and inferring rules from the results of these experiments*". He argues that they will understand the domain at a higher level than when the necessary information is just presented by a teacher or an expository learning environment. In most cases, discover learning is tied to problem-solving, Borthick and Jones [5] write that "*participants learn to recognize a problem, characterize what a solution would look like, search for relevant information, develop a solution strategy, and execute the chosen strategy*".

For us, a type of discovery learning called problem-based learning is particularly interesting, which is defined by Finkle and Torp [6] as "*a curriculum development and instructional system that simultaneously develops both problem-solving strategies and disciplinary knowledge bases and skills by placing students in the active role of problem solvers confronted with an ill-structured problem that mirrors real-world problems.*" Our approach is very close to this definition, as we will show it below.

1.2. Mathematics camps in Hungary

Mathematics talent education has a strong tradition in Hungary, and there are numerous mathematics camps. Here we describe the camps organized by The Joy of Thinking foundation [7], established by Lajos Pósa. These weekend math camps are characterized by the internationally renowned Pósa-method, which is a form of guided discovery learning. The author, being an ex-student of Lajos Pósa, has been assisting in these camps for many years, and now teaching two groups since 2014, so he has a working knowledge of Pósa's pedagogy.

According to Bibergall [8], guided discovery learning is characterized by convergent thinking. "*The educator devises a series of statements or questions that guide the learner step by step, making a series of discoveries that leads to a predetermined goal*". In our math camps, the learner is guided through exercises that have strong interconnection under the surface. Katona and Szűcs [9] describe this as a web of problem threads.

1.3. Problem threads in the Pósa-method

A problem thread consists of tasks that have a connection, which can be of different types. A type of connection might be that they share the topic, e.g. graph theory. Another type of connection is when the problems build on top of each other, meaning that the solution of one task needs certain ideas, methods that are more easily available for students if they solved a previous task. This common element of thinking, which links tasks in one thread, is called the kernel of the thread by Katona and Szűcs [9].

For example, the above-mentioned kernel could be induction, which Pósa calls chain-reaction for young students and starts with a simple logical task of the style "who robbed the bank", and later on students will get to proving complex theorems like "every tournament graph has a Hamiltonian path" using induction. Here we would like to mention that the latter statement is not presented in such a plain way, but instead, an open question with dragons carrying people between islands, to make it more fun for kids.

Pósa always emphasizes not to provide a statement to prove, but ask an open question instead, or even better – which happens in this dragon-world – just present a situation and let the students ask questions. We aim to introduce kids to research, and particularly in mathematics an interesting question is very valuable to the scientific community. In this regard, the Pósa-method is also a type of inquiry-based learning. Students start with divergent thinking when solving a task, the experimentation in the domain of mathematics has a significant role in Pósa's pedagogy.

1.4. The web of problem threads

During the mathematics sessions of the previously mentioned camps, there are always multiple problem threads running in parallel, which means that a lot of tasks from different threads are presented to the students simultaneously. The threads are not isolated, they may have meeting points, common problems, they may have important links or dependencies between them, forming a web of problem threads.

The web of problem threads is like a master plan, leading the learner to acquire knowledge and skills that are our educational goals. So, the main challenge of the teacher is to design the curriculum to suit the intended development of students, which means identifying the competencies to learn, organizing them in the right system, and collecting or creating a vast amount of problems and exercises that will trigger and guide the learning process. In the following, we show a system of topics and methods accompanied by exercises that serve as a base of our web of problem threads for algorithmic programming talent education.

2. Discovery learning in algorithmic programming

We need to define our educational goals. As for the mathematics talent education program, Juhász [10] says "*children should be taught how to think, rather than making them learn theorems and formulas by heart or giving them ready-made methods to solve problems*". Following this principle, our focus is on teaching algorithmic thinking and problem-solving. Another important objective is to show the joy in thinking about interesting problems and creating working programs to solve them. With this, we would like to open up the world of competitive programming for the children.

The emphasis is not on competition, but these contests are aimed to test the algorithmic thinking and problem-solving skills of the participants with "nice" tasks. The community of qualified programmers is preparing the problems of these competitions and they make them so that other people would enjoy thinking about them. There is a certain beauty in problems that is hard to describe,

and it is much celebrated within the community. This beauty can come from an interesting question, an elegant solution, application of a method in an unexpected situation, a nice idea, the connection between different topics, etc. So, the world of competitive programming is partly self-serving, it provides fun for people doing it, very much like how Lajos Pósa describes the world of mathematics [11].

Computer programming is a bit different from mathematics. There are a lot of standard algorithms and data structures that are almost ready-made methods that you need to customize, combine, and apply in numerous different scenarios. We try to teach them through a series of problems, having the students discover them mostly on their own, if possible. However, we put more emphasis on the applications of these methods in different problems. Therefore, we consider our approach a problem-based pedagogy. The problems have similar dependencies and connections between each other as the ones in Pósa's mathematics camps. We create problem threads for the algorithms and data structures we teach and try to connect them, thus forming our web of problems. Fortunately, the tasks at programming contests usually have some funny stories to cover the underlying problem, so at first sight it is not obvious to which thread they belong.

2.1. The objectives of our curriculum, related work

Designing the curriculum starts with identifying the topics and methods we want to teach. Programming competitions reflect quite well what the community of computer scientists consider important knowledge and skills in the field of algorithmic programming. We selected the elements of this curriculum by looking at materials of competitions and those, up-to-date literature helping to prepare for contests.

Our most important source is the Syllabus for the International Olympiad in Informatics (IOI) [12], it provides an excellent summary of expected knowledge. Competitive Programming 3 by Halim et al. [2] contains most of the topics occurring at ACM International Collegiate Programming Contest for university students. We examined the contents of Laaksonen's Competitive Programmer's Handbook as well [13]. There are some excellent on-line resources and tutorials collecting the important algorithms and data structures, for example the CP-Algorithms website [14] Geeks for Geeks [15] and various blogposts on Codeforces [16-17]. Using these sources, and our experience in the history of high school competitions, we compiled the contents of our curriculum, which are the most relevant knowledge for high school students in our opinion.

There are numerous similar articles, especially about the topic of how to prepare students for competitions. Király [18] describes a whole roadmap of teaching programming from the very beginning to the preparations for the IOI, together with proper pedagogical guidelines and useful advice. In her doctoral dissertation, Erdősné [19] provides a detailed insight into Hungarian and international talent education in informatics, also outlining a plan of teaching advanced level programming throughout secondary school. At the concrete topics where they include tasks, both papers present very similar exercises to our chosen ones. In comparison to their work and the above-mentioned literature, the novelty of this curriculum lies in tailoring the system of problems to the discovery learning method of Pósa. Since the selection of tasks and the interconnections play a crucial part in our didactics, we provide a more detailed and more complete list of exercises, organized in the structure described below.

2.2 Design and overview of the curriculum

Most of the books, online courses and tutorials about programming focus on the elements of the language. We have different goals, so these are out of scope for us, but we would like to build on the basics. Teaching algorithms, we assume knowledge of language elements, like variables, types, operators, conditional statements, loops, etc. We start at the very basic algorithmic structures and

finish with some of the most complex methods required for the IOI. The primary use of this curriculum will be a talent education program with groups of gifted children beginning around age 12 and lasting until age 17. The examples of Pósa's mathematics camps showed us that it is possible the keep such groups together for years, develop their knowledge systematically, and teach topics building upon each other.

For our problem-based method, the curriculum would be a huge network of tasks, through which students discover, practice and expand their knowledge. To design it, first, we look at the bigger picture, organize the theoretical backgrounds of the tasks, and identifying interconnections between them. The resources we used, which are mentioned above in section 2.1, present the materials categorized and ordered by topics, which is great if we search for something we already know, but not the ideal order for learning. We not only talk about their order of difficulty, but our goal is also to find the ideal plan, where we minimize the number of great ideas needed to work out the desired algorithms.

Figure 1 shows a graph of curriculum modules, each abbreviated by a short code for readability. Full names and descriptions of these can be found in the tables of section 3. We categorized these modules into four types that have different colors on the graph:

- problem-solving techniques (red),
- algorithms or algorithm templates (blue),
- data structures (green),
- theoretical backgrounds or subjects (yellow).

Interconnections between these elements are visualized by arrows, the thicker the arrow is, the stronger we find the dependency between them. We also use light grey arrows, which don't mean dependency, rather similarities, when knowing a method is helpful for the other one.

Later, in section 3 we divide the modules to units in order to have reviewable segments. We provide a collection of problems for each unit, dedicated to modules, with certain objectives. This collection is not complete, and never meant to be, extending and changing it constantly is part of our philosophy. These problems serve as a good skeleton for starting a talent education program with discovery learning. For our pedagogy, we define the following three important types of tasks.

- Introduction problems are the very first problems of a topic that can induce discovery.
- Reinforcement problems involve the application of a previously discovered method in new situations, or they can be practice problems as well, their goal is to strengthen students' knowledge.
- Synthesis problems in our terminology mean tasks that bring together multiple learned methods or require a high level of understanding of the concepts involved.

Problems are, of course, very often linked to topics and problems in different units, which cannot be shown in the format of this document.

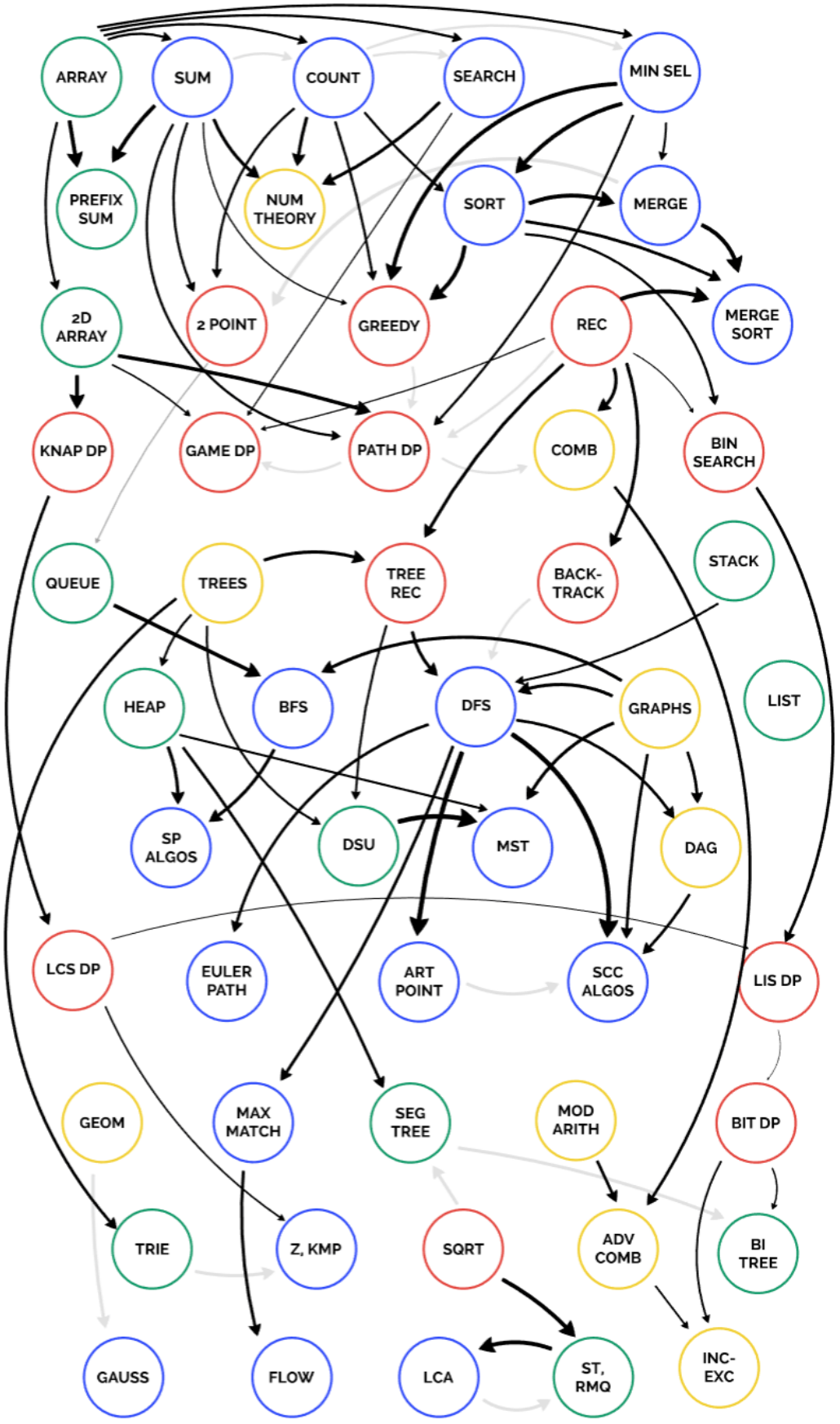

Figure 1: The graph of the curriculum

3. Details of the curriculum

Below we present the parts of the curriculum with more detailed descriptions. Grouping them into units serves as an aid for presenting, there are no hard boundaries between them, and there are very important interconnections in-between units. We estimate the length of the units to take up 2-3 weekend camps, which can be scheduled in one year or even in one semester.

Within units, we propose problems for teaching the algorithms and data structures, where applicable. They are collected from various online problem sets that are publicly available and have a judgement system to verify solutions. This is important for us, we would like to show the students that not only the theory but the implementation is an essential part of computer science. We use identifiers for the problems, that consist of an abbreviation of the problem source listed below, and a handle of the problem, which can be used to find it (by web search or on the site itself).

- CF: Codeforces, CFG: Codeforces Gym [3].
- HR: HackerRank [24].
- CC: CodeChef [25].
- CSA: CS Academy [26].
- SPOJ: Sphere Online Judge [27].
- UVa: UVa Online Judge [28].
- IOI: Tasks of the International Olympiad in Informatics. Grader for most of them can be found at PEG Online Judge [29].
- M: Mester [30]. A Hungarian problem collection. Unfortunately, translations for these problems are not available yet, we plan to write them for the ones that we use, in a new system that is under development at Eötvös Loránd University. (We give the English translation of task names in parentheses.)
- ICPC: ICPC Live Archive [31].
- TIMUS: Timus Online Judge [32].

Erdősné [20] gives an excellent overview of online judges and their features that includes all the above-listed websites. As the reader can see in the following chapters, in our problem collection we favor Codeforces, HackerRank, and Mester, because using practice mode in these the student (or the teacher) has access to the small test cases and their expected answers, which is very helpful for debugging.

3.1. Unit 1: Introduction

In this unit, we introduce some basic algorithmic patterns that are necessary as building blocks of further, more complex algorithms. Most of them operate on a sequence of numbers, so we heavily use the one-dimensional array data structure. Zsakó and Szlávi refer to them as programming theorems [27]. Students need to acquire firm knowledge of these basics so that later they can easily apply them as part of a compound solution. Knowing the required language elements, talented students can invent algorithms to solve these tasks. While there is no inevitable dependency between them, we recommend introducing them in the order below. As the first combination of loops and conditional statements (*for* and *if*), counting elements with a certain property (e.g. even numbers) is a straightforward problem.

For introducing the algorithms listed below, we generally use the tasks available in Mester [30], the Hungarian problem set, categorized accordingly, so we don't list them here. It is also not difficult to invent our own tasks for these basics. Number theory is a great topic for applying and combining what we learned in this unit. For example, deciding whether a number is a prime comes down to searching for a divisor of it. With summing the divisors, one can search for amicable numbers. It

is also a great opportunity to introduce functions as a language element. Discovering the Euclidean algorithm, students can see something advanced and very elegant. There is a great task guiding to the Euclidean algorithm (*CF - 527A. Playing with Paper*), in which you start with a rectangular paper, and always cut off square-shaped parts of it.

Code	Name	Description
SUM	Sum	Calculate the sum of a series of numbers.
COUNT	Count	Count the elements of a series with a certain property.
SEARCH	Search	Search for the element(s) with a given property in a sequence.
MIN SEL	Minimum Selection	Select the minimum/maximum in a sequence.
SORT	Sort	Sort a sequence with simple algorithms, like bubble sort, min. selection sort, counting sort, etc.
MERGE	Merge	Compute the intersection/union of two sets (ordered list of data), using the linear merge algorithm.
NUM THEORY	Basic Number Theory	Calculate the number of divisors and sum of divisors of an integer, search for primes, prime factorization, Euclidean algorithm for GCD.

Table 1: Modules of the first unit

3.2. Unit 2: Some basic problem-solving techniques

In this unit, we dive into the world of programming contest problems. Computing solutions quickly becomes a key factor, we introduce the notion of computational complexity and analyze every solution from this aspect. Greedy algorithms with obvious greedy decisions are a great start to provide easy success to everyone, and not much later problems for dynamic programming will show that greedy doesn't always work. The technique of recursion is essential to introduce early since it is used in multiple other methods of this unit. Building on the concept of recursion, and knowledge from unit 1, we can guide the learners to efficient sorting algorithms, like merge sort and quicksort.

Code	Name	Description
GREEDY	Greedy Algorithms	Solve problems using greedy decisions, recognize whether it leads to the optimum.
REC	Recursion	Get familiar with the power of recursion in typical scenarios.
MERGE SORT	Merge Sort	Recursive sort algorithms: merge sort and quicksort.
2 POINT	Two Pointers	The two pointers principle for speeding up some optimization tasks.
PREFIX SUM	Prefix Sum	The prefix sum / cumulative sum of a sequence, as a data structure.
PATH DP	DP for Finding Best Path	Introduction to dynamic programming: optimize a route on a grid.
GAME DP	DP for Simple Games	Find the winning strategy in simple two-player games with dynamic programming.
COMB	Combinatorics	Basic tasks involving combinatorics, like permutations, combinations, Fibonacci-type sequences, etc.
BIN SEARCH	Binary search	Bisect to find a value in a sorted range, and to find extremum using a predicate.
BACKTRACK	Backtrack	Speed up brute-force algorithms by backtracking, 8 queens' problem and similar.
KNAP DP	DP in Knapsack problem	Some typical DP problems: Coin Change, Knapsack and alike.

Table 2: Modules of the second unit

With the prefix sum data structure, students can construct their first powerful data structure to answer queries. Binary search appears in the form of looking for a value in a sorted range, and then comes the great idea to use it when maximizing or minimizing some target with some constraints (e.g. in task *CF - 760B. Frodo and Pillows*). We also show problems where only exponential solutions are known, but we can speed them up using backtracking. A nice introduction task could be to generate all balanced parentheses sequences of a certain length.

Dynamic programming could be built up in different ways, we propose starting it with problems in which we examine paths on a grid with only right and downwards steps. In this case, calculating partial results with a bottom-up strategy comes as a natural idea, much easier conceptually than transforming recursive formulas in problems like the Coin Change or the Knapsack problem. We eventually get there as well, but in the meantime, we also take a step applying dynamic programming for computing winning strategies in simple two-player games. It is used also in combinatorial problems, which are at this level mostly related to the Pascal triangle, Fibonacci-type sequences, permutations and variations. Below we include a table showing example tasks that we propose for the modules in this unit, according to the principles described in section 2.

Code	Introduction	Reinforcement	Synthesis
GREEDY	M - Wifi, HR - Priyanka and Toys	M - Mekk Elek (Mekk Elek the Handyman) M - Fénykép (Photo) CF - 349B. Color the Fence	M - Termek (Rooms) CF - 1077E. Thematic Contests
REC	Towers of Hanoi	M - Felbontás (Decomposition)	HR - The power sum
2 POINT	CF - 660C. Hard problem, M - Autószállítás (Car Shipping), CF - 1133C. Balanced Team	CF - 616D. Longest k-good Segment, CF - 1006C. Three Parts of the Array	M - Nyaralások (Trips), IOI11 - Rice Hub
PREFIX SUM	SPOJ - CSUMQ, CF - 313B. Ilya and Queries,	M - Távoli bolygó (Distant Planet), CF - 816B. Karen and Coffee	M - Képátló (Image Diagonal), IOI11 - Rice Hub
PATH DP	M - Kincsek a hegyoldalon (Treasures on the hillside), M - Pontgyűjtő verseny (Point collecting contest)	M - Benzin (Gasoline), CF - 429B. Working out	M - Lépcsők (Stairs), M - Képátló (Image diagonal), CF - 407B. Long Path
GAME DP	HR - Game of Stones	HR - A Chessboard Game, M - Számok elvétele (Removing Numbers)	M - Fehér és Fekete korongok (White and Black Tokens), CF - 731E. Funny Game
COMB	HR - Picking Cards, CF - 617B. Chocolate	HR - Sherlock and Pairs, CF - 894A. QAQ	M - Lépcsők (Stairs), HR - Merge List
BIN SEARCH	CF - 706B. Interesting Drink, CF - 600B. Queries About Less or Equal Elements	CF - 760B. Frodo and Pillows, CF - 670D2. Magic Powder	IOI11 - Rice Hub, UVa - 1079. A Careful Approach
BACKTRACK	Balanced Parentheses, 8 Queens problem	M - Ültetés (Seating)	CC - KOL1510
KNAP DP	M - Nem kifizethető címlet (Unpayable Amount), M - Bélyeg (Stamps), SPOJ - KNAPSACK	CF - 19B. Checkout Assistant, M - Munkagépek (Machines), M - Vásár (Sale)	CFG - 102534B. Need More T-shirts, CF - 1132E. Knapsack

Table 3: Problems for the second unit

3.3. Unit 3: Graph theory-driven problems and algorithms

A lot of real-life problems can be formulated with graphs, and so they appear frequently in programming competitions above a certain level. In this unit, we teach the most commonly used, but still not too complex algorithms. Some theory is involved, students need to learn the notion of trees and graphs. We recommend problems on rooted trees first which have recursive solutions, because they are quite elegant and serve as a base for depth-first search. Three basic, linear data structures can be learned concurrently, stack, queue and double-ended queue, we apply them in graph traversals, and at this point some beautiful and hard problems can show their advantages (*HR - Largest Rectangle*, *HR - Deque-STL*).

The two types of graph traversal (breadth-first and depth-first) are introduced on simple, undirected graphs. At this level, breadth-first search has more applications, while we build on depth-first search a lot in the next unit. The graph traversals work the same way in directed graphs, and particularly acyclic graphs of this type are interesting for us, they model practical problems like scheduling a project, university studies or this curriculum itself. There is a nice combinatorics task, where the question is how many different paths are between two vertices (*HR - Kingdom Connectivity*).

Code	Name	Description
TREES	Trees, Binary Trees	Tree structure appearing in different situations, e.g. family tree, company structure.
TREE REC	Recursion on trees	Compute values for trees using recursion, problems involving a hierarchical structure.
LIST	Linked lists	Know the basics of linked data structures, and when to use them considering their advantages and disadvantages.
STACK	Stack	Understand and use the stack (LIFO) data structure.
QUEUE	Queue, Deque	Understand and use the queue (FIFO) and double-ended queue data structures.
GRAPHS	Graphs	Conceptual introduction of graphs as a background of different problems.
BFS	Breadth-First Search	Solving problems using graph traversal in increasing order of distance from a vertex.
DFS	Depth First Search	The recursive depth-first search algorithm and basic applications.
DAG	Directed Acyclic Graphs	Problems involving a DAG, like critical path method, topological ordering.
SP ALGOS	Shortest Path Algorithms	Find shortest paths in a graph. Bellman-Ford, Floyd-Warshall, Dijkstra algorithms.
HEAP	Heap, Priority Queue	Understand the heap data structure, and use priority queue when needed, e.g. in Dijkstra and Prim algorithms
MST	Minimum Spanning Tree	Find the minimum spanning tree in graphs. Kruskal and Prim algorithms.
DSU	Disjoint Set Union	The DSU (or Union-find) data structure applied in various problems, e.g. Kruskal algorithm.

Table 4: Modules of the third unit

Two data structures, DSU and heap are included here for two reasons: they are necessary for efficient implementations of Kruskal, Dijkstra and Prim algorithms, and both are viewed as rooted trees, so they perfectly fit in here. In programming competitions, heap is generally applied by using the priority queue included in standard libraries, while DSU needs to be implemented.

We conclude the unit with two more advanced problems on weighted graphs: shortest paths and minimum spanning trees. Well-known algorithms listed below can be discovered by students with some hints. Since weighted graphs can model various real-world problems, there are a huge amount of competition tasks where these algorithms are necessary with some modifications.

Code	Introduction	Reinforcement	Synthesis
TREE REC	M - Titkos társaság (Secret association)	CF - 115A. Party, CF - 580C. Kefa and Park	HR - Even Tree
STACK	HR - Equal Stacks, UVa - 514. Rails	HR - Balanced Brackets	HR - Largest Rectangle CF - 547B. Mike and Feet
QUEUE	UVa - 10935. Throwing cards away	HR - Deque-STL	IOI06 - Pyramid
BFS	List vertices in order of distance from one vertex	M - Randi (Date), M - Csapat (Team), CF - 796D. Police Stations	M - Mérőkannák (Measuring cups), CSA - BFS-DFS
DFS	SPOJ - ABCPATH	CF - 445B DZY Loves Chemistry, M - Utcaseprő (Street sweeper)	CF - 1316D. Nash Matrix CSA - BFS-DFS
DAG	M - Építkezés (Contruction), M - Utak száma (Number of routes)	CF- 915D. Almost Acyclic Graph CF - 512A. Fox and Names	HR - Kingdom Connectivity
SP ALGOS	CF - 20C. Dijkstra, CF - 295B. Greg and Graph	M - Autóbusz járatok (Bus lines), HR - Jack goes to Rapture	M - Telephelyek (Sites), IOI11 - Crocodile
MST	SPOJ - MST	M - Malom (Mill)	HR - Roads in Hackerland CF - 160D. Edges in MST
DSU	CF - 1095F. Make It Connected	M - Hálózat tesztelés SPOJ - CONSEC	CC - ABROADS CF - 875F. Royal Questions

Table 5: Problems for the third unit

3.4. Unit 4: Various advanced algorithms, geometry, combinatorics

Our fourth unit contains some theoretically complicated algorithms. Speaking of Hungarian national contests, these are only required for the highest age group (11-12th grade). One can argue that discovery learning is not possible in some of these topics. We still aim for introducing them through exercises in which we provide hints to the students. Furthermore, deep understanding can be also achieved when the students implement solutions based on these complex algorithms.

Code	Name	Description
LCS DP	Longest Common Subsequence	Dynamic programming using non-trivial two-dimensional arrays.
LIS DP	Longest Increasing Subsequence	Dynamic programming sped up with binary search or other methods.
BIT DP	Bitmask DP	Dynamic programming on subsets, using the bit vector representation of sets.
ART POINT	Articulation Points, Bridges	Biconnected graphs, Tarjan's algorithm and the L-value for finding cut vertices and edges.
SCC ALGOS	Strongly Connected Components	Kosaraju's and Tarjan's algorithm and applications.
EULER PATH	Eulerian Path	Condition of Eulerian path or circuit and finding it in directed and undirected graphs.
MAX MATCH	Maximal Matching	Hungarian algorithm for the maximal matching in a bipartite graph.
MOD ARITH	Modular Arithmetics	Calculate powers and inverses efficiently modulo a given number.
ADV COMB	Advanced Combinatorics	Various difficult combinatorics problems.
INC-EXC	Inclusion-Exclusion Principle	Apply the inclusion-exclusion principle to answer some questions in combinatorics.
GEOM	Geometry	Geometric problems on the Cartesian plane.

Table 6: Modules of the fourth unit

There are graph theory topics (Articulation Points and Bridges, Strongly Connected Components, Eulerian Path, Maximal Matching) which mostly rely on depth-first search and its extensions. We would like to mention two beautiful tasks, *CF - 508D. Tanya and Password*, which is a surprising application of Eulerian paths, and *UVa - 12668. Attacking Rooks*, where the idea is to introduce a bipartite graph where the edges are fields on the chess table.

Dynamic programming is present throughout the curriculum in many other algorithms, but here we revisit it with more advanced applications. Below, we named two characteristic tasks (Longest Common Subsequence and Longest Increasing Subsequence), but there are much more included, in the first topic could be any other task that requires a non-trivial two-dimensional array formulation, and the second is related to tasks where we combine dynamic programming with other techniques, like binary search in the example problem.

Advanced Combinatorics includes not only questions about how many ways we can construct something, there is often an ordering (e.g. lexicographical) defined between these and telling the element at a given position requires a profound understanding of recursive patterns. The number of solutions is often very large, and then the answer is expected modulo some big prime, so the apparatus of modular arithmetic is used here. That is also very interesting itself, cryptographic applications can be visited.

Code	Introduction	Reinforcement	Synthesis
LCS DP	HR - The LCS	M - Jelek (Signs), M - Rúd felvágás (Stick cutting)	CF - 607B. Zuma, HR - LCS Returns IOI09 - Raisins
LIS DP	M - Konténeroszlopok (Container Columns)	M - Kockákból legmagasabb torony (Highest Tower of Cubes)	CF - 650D. Zip-line
BIT DP	M - Vásárlások (Purchases)	CF - 580D. Kefa and Dishes	CFG - 102128B. Cake Tasting
ART POINT	SPOJ - SUBMERGE	M - Duplán elérhető pontok (Double reachable points)	CF - 700C. Break Up CF - 732F. Tourist Reform
SCC ALGOS	SPOJ - CAPCITY UVa 13057 - Prove Them All	CF - 427C. Checkposts CF- 949 C. Data Center Maintenance	M - Hercegek házassága (Wedding of Princes) SPOJ - ADAPANEL
EULER PATH	M - Zárkód (Lock code) CF - 1334D. Minimum Euler Cycle	CF - 508D. Tanya and Password	M - Dominó
MAX MATCH	HR - Real Estate Broker	UVa - 12668. Attacking Rooks CF - 498C. Array and Operations	M - Hercegek házassága (Wedding of Princes) SPOJ - QUEST4
MOD ARITH	UVa - 10104. Euclid Problem	CF - 300C. Beautiful Numbers, CF - 717A. Festival Organization	M - Szigetek (Islands), HR - Game of Thrones II
ADV COMB	HR - Lexicographic steps, CF - 9D. How many Trees? CC - NWAYS	CF - 612E. Square Root of Permutation, M - Birtokfelosztás (Dividing land)	M - Szigetek (Islands), HR - Game of Thrones II
INC-EXC	SPOJ - NGM2	UVa - 11806. Cheerleaders	CF - 102128B. Cake Tasting
GEOM	M - Házak (Houses) M - Zárt poligon készítése (Creating a closed polygon)	M - Autópálya (Highway) M - Háromszög (Triangle)	M - Hegy (Mountain) CF - 552D. Vanya and Triangles. UVa - 12278. 3-sided dice

Table 7: Problems for the fourth unit

Geometry is a huge category, it starts with basic operations, like computing orientations, deciding if segments intersect, etc. and leads to sophisticated methods like sweep-line principle and convex hull algorithm. It is a very good example of how we can build up a topic step-by-step. We need to rely on various knowledge from mathematics at school, most importantly the Cartesian coordinate system.

3.5. Unit 5: Complex data structures, string algorithms

The final unit is dominated by data structures and contains some topics which are not required even for the IOI. Teaching data structures like Segment Tree, Fenwick Tree, Trie or Sparse Table with discovery learning is quite difficult, we have not researched this area extensively yet. Currently our pedagogy goes with describing and visualizing them on examples, and having the students work out the implementation for deeper understanding. The emphasis is on customizing them and using them in various new scenarios. A great example is using the Trie for binary numbers, e.g. finding the pair of numbers with maximal XOR value in a given set.

We also aim to capacitate the students to describe the data structures that they need to solve a certain problem, in terms of its operations and their maximum complexity. The next step is designing such a data structure, which is usually adapting a known data structure appropriately.

Code	Name	Description
SEG TREE	Segment Tree	The Segment Tree data structure for updating and querying certain computed values in a range.
BI TREE	Binary Indexed / Fenwick Tree	The Fenwick Tree data structure as an alternative to segment trees.
TRIE	Trie, Suffix Tree, Suffix Array	Data structures for storing and searching text corpora: Trie, Suffix Tree, Suffix Array.
Z, KMP	Z-algorithm, Knuth-Morris-Pratt	Advanced string pattern matching algorithms: Z, KMP algorithm.
FLOW	Network Flows	Modell problems as network flows, minimum cut maximum flow algorithm.
GAUSS	Gaussian Elimination	Solve a system of linear equations.
LCA	Lowest Common Ancestor	Finding the lowest common ancestor of tree vertices, and its applications.
SQRT	SQRT Decomposition	The Square Root Decomposition problem-solving technique. Mo's algorithm.
ST, RMQ	Sparse Table, Range Minimum Query	Sparse Table for solving the Range Minimum Query problem, plus further applications.

Table 8: Modules of the fifth unit

The unit also contains the topic of Network Flows, solving a system of linear equations by Gaussian elimination, and Square Root Decomposition as a problem-solving principle. All of them appear in some very hard tasks, with tricky applications. Computing the Lowest Common Ancestor and Range Minimum Query are connected topics, furthermore, they can be related to data structures in this unit.

Finally, we included some advanced string processing algorithms in this unit, which are mostly about efficiently searching a pattern in a text. However, they can be used for many different problems, like counting the distinct substrings of a string. Together with the Trie, Suffix Tree and Suffix Array data structures, they form a whole toolset to tackle tasks with strings.

Code	Introduction	Reinforcement	Synthesis
SEG TREE	UVa - 12532. Interval Product	CF - 380C. Sereja and Brackets, CF - 474F. Ant Colony	CF - 524E. Rooks and Rectangles CF - 242E. XOR on Segment
BI TREE	SPOJ - INVCNT	CF - 61E. Enemy is Weak	SPOJ - DCEPC206
TRIE	SPOJ - PHONELST	CF - 455B. A Lot of Games	ICPC - 4682. XOR Sum
Z, KMP	CC - KAN13C	CC - TASHIFT, CF - 126B. Password	SPOJ - DISUBSTR
FLOW	SPOJ - POTHOLE	UVa - 820. Internet Bandwidth	CF - 546E. Soldier and Traveling
GAUSS	SPOJ - XMAX	TIMUS - 1042. Central Heating	
LCA	HR - Kth Ancestor	TIMUS - 1471. Distance in the Tree	CF - 342E. Xenia and Tree
SQRT	SPOJ - GIVEAWAY	CF - 13E. Holes, CF - 86D. Powerful Array	CF - 342E. Xenia and Tree

Table 9: Problems for the fifth unit

4. Following a topic throughout the curriculum: Dynamic Programming

4.1. Our approach to DP

Many educational materials introduce Dynamic Programming (DP) using recursion, as a way of overcoming the time complexity caused by the curse of recursion. We would like to show an alternative way here, without claiming any of them better. Erdősné [21] and Forišek [22] give an excellent overview of the place of DP in popular algorithm textbooks, and rightfully argue that those books do not provide a good approach to present the DP paradigm to secondary school students. Both papers suggest teaching DP after recursion, Forišek [22] even gives a strategy to transform a top-down recursive solution to a bottom-up DP solution. Király [18] suggests that DP should be taught only after recursion and backtracking and greedy algorithms at an advanced stage of programming knowledge. Independently from the three mentioned authors, we selected very similar problems and almost the same order of them. There is one crucial difference: our approach starts with the bottom-up strategy. We agree with Erdősné [21] that the LOGO language can provide very solid grounds for recursion at a young age, but unfortunately in recent years we see a lot of children who start learning algorithmic programming without doing any LOGO before. This is one reason why we do not wish to rely on recursion.

In our scenario, there are young kids, for whom arrays are generally easy to understand, while functions are harder, recursive functions even more. Since we teach programming and algorithms together, and we consider the features of the programming language as tools for our algorithms, we can bring up the concept of DP with very simple problems, even before teaching functions. Regarding the approach of Forišek [22], another aspect of our method is that we do not start with presenting a method of problem-solving, but give a problem to the children, in which the single way of succeeding is the solution that we intend to teach. For this, we need problems where they don't have another choice, but to follow the way the teacher wants to follow. If we started with Fibonacci numbers, expecting children to first come up with the recursive solution and then figure out the steps to convert it to a bottom-up DP, our educational goals might be "screwed" by smart kids, who immediately solve it with an array - which is quite a natural solution.

We aim to introduce DP as a way of solving a task by "filling a table". It sounds much more innocent than "decomposing to subproblems", even though that is happening under the surface. We will include some formulas to show how the DP tasks get more and more complex as we advance in the curriculum.

4.2. First steps

A simple introductory task is listed above as *M - Kincsek a hegyoldalon (Treasures on the hillside)*, in which there are treasures on a grid, and we need to collect as many as we can, only moving to the right and down, starting from the upper left corner. The idea of calculating the maximum amount of treasures we can take to each cell comes naturally. If not, our pedagogy involves being ready to give good hints. In this case, we usually present a complicated example on paper and ask the students to solve it by hand, during which they most likely come up with the desired method.

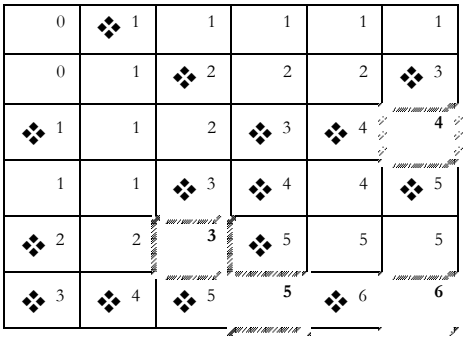

Figure 2: A concrete example of calculating the most collectable treasures for each cell

The formula which lies beneath this problem goes as follows:

```
DP[i,j] = max(DP[i-1,j], DP[i,j-1]) + T[i,j]        (1)
DP[i,0] = 0, DP[0,j] = 0
```

For practice, we use another "collect points on a grid" type exercise, *M - Pontgyűjtő verseny (Point collecting contest)*, where the possible movements are different. As a reinforcement, we propose a similar, but a bit more difficult task, *M - Benzin (Gasoline)*, which also includes constructing the optimal path. A quite hard task belonging to this group is *CF - 429B. Working out*, it can be used later to refresh the knowledge. We called this category PATH DP, expressing that we are usually searching for an optimal path to the destination.

Starting a bit later, but parallel with this thread, we analyze simple two-player combinatorial games and construct optimal strategies by determining the winning and losing positions. This is also a great opportunity to start the exploration of this world offline, with actually playing some simple games and finding their winning strategy without a computer.

A very basic problem is *HR - Game of Stones*, where two players take away 2, 3 or 5 stones from a pile in alternating turns, and the one unable to take, loses. The task is to tell who will win starting from various number of stones if both players play optimally. The problem could be generalized with different allowed moves. The solution programmatically comes down to deciding for every `i` number of stones increasingly, whether it is "good" to leave `i` stones, based upon that we know the previous answers. The notion of winning and losing states and their properties can be formulated at this point. The formula here would look like this:

```
DP[i] = not(DP[i-2] or DP[i-3] or DP[i-5])          (2)
DP at negative values treated as false
```

Our reinforcement task in this topic is a conceptually simple, but programmatically complex, two-dimensional game played on the chessboard, *HR - A Chessboard Game*. In this game both players move one token taking turns, they can make the knight moves that decrease the sum of coordinates. With this, the dynamic programming nature of game analysis becomes clear, and students also face a problem, where the order of computing the DP table elements is not straightforward. If they have a firm understanding of recursion, we can show them the power of recursion with memorization (called memoization). The below expression describes the computation in this task:

```
DP[i,j] = not(DP[i-2,j+1] or DP[i-2,j-1] or        (3)
              DP[i-1,j-2] or DP[i+1,j-2])
DP outside the table treated as false
```

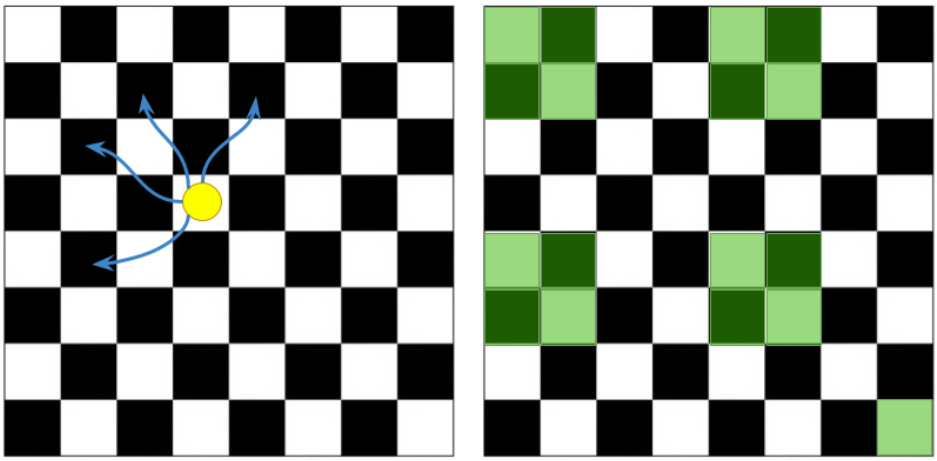

Figure 3: The moves and winning fields of the chessboard game

The synthesis problems for this module should occur much later, but we want to mention here that the task *M - Fehér és Fekete korongok (White and Black Tokens)*. There is a row of white and black tokens in this game, a player in one move can take a token from either the beginning, or the end of the row, and both players have to maximize their white tokens. This task is a great opportunity to first come up with a solution that constructs a 2D array, where the problem itself involves only a sequence. Here the states of the game guide us to the 2D data structure, and this motif is very important in the more advanced tasks of the LCS DP module.

There is an important connection with the Combinatorics module of this unit, namely that DP is often the method to solve a combinatorics problem. Calculating elements of the Pascal-triangle can be viewed as a DP task as well. An excellent problem involving a 1D array filling is *M - Lépcsők (Stairs)*. The question is how many ways you can go up N stairs if you can take steps of at most K stairs. We suggest scheduling this task soon after the first DP problems, in parallel with other PATH DP tasks. The bottom-up nature of DP is very clearly visible, as we count the ways of reaching each stair in order.

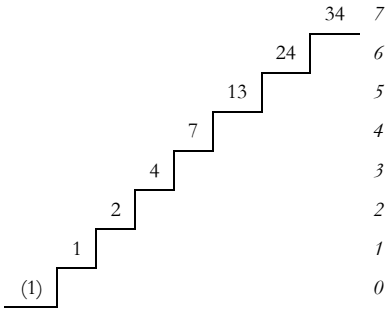

Figure 4: The number of ways to reach each stair, with maximal step size K=3

The formula of the solution is the first of this kind that doesn't have a closed form:

```
DP[i] = DP[i-1] + DP[i-2] + … + DP[i-K]          (4)
DP[0] = 1, DP at negative values treated as 0
```

Besides DP, we plan to introduce recursion approximately at the same time, or just a little bit shifted. The close relation between these strategies should be enlightened and demonstrated thoroughly. The two topics running in parallel helps to strengthen their grounds. Furthermore, DP in

some cases can be more elegantly performed using recursion with memoization, particularly when the computing order is not trivial.

4.3. Exploring the power of DP

After feeling the taste of DP with problems where transitions are steps in a game-like scenario, the students will meet less obvious DP problems in different wrapping. We named this module Knapsack DP after the very representative Knapsack problem.

We start with a similar, but much easier task, the Coin Change problem, where the question is what amounts of money can be paid using some set of banknotes (not asking for the minimum number of notes yet). It can be modeled with a single 1D boolean array, and deciding payability for each integer amount increasingly is an intuitive idea (programmatically very similar to *HR - Game of Stones*). With `B[1..N]` denoting the values of coins, we can formulate the solution as follows:

```
DP[i] = DP[i-B[1]] or DP[i-B[2]] or … or          (5)
        DP[i-B[N]]
DP[0] = true
DP at negative values treated as false
```

Minimizing the number of notes is the next step and it is a representative example of greedy not working. The *M - Bélyeg (Stamps)* task is essentially this problem.

The Knapsack problem, *SPOJ - KNAPSACK* has a special role in our DP curriculum, this is the first time when we use a non-trivial task decomposition with two variables. There are `N` items with different sizes and values, and we have to fit as much as we can into a backpack with size `K`. We calculate the maximum value for each backpack size when considering the inclusion of the items one by one. Here we present the underlying formula, where `V[1..N]` are the values, and `S[1..N]` are the sizes of the items:

```
DP[i,j]= max(DP[i-1,j], DP[i-1,j-S[i]] + V[i])    (6)
DP[i,0] = 0, DP[0,j] = 0
DP at negative values treated as -∞
```

Many other real-life problems can be solved with this method. The two reinforcement tasks we suggest are such examples. In *M - Munkagépek (Machines),* distributing jobs between two machines can be reformulated to a special coin change problem. In the *M - Vásár (Sale)* problem, we have to maximize the profit of a merchant, and it can be reduced to a Knapsack problem.

The knowledge of DP is essential in some graph algorithms in our third unit. Two very common shortest path algorithms, Bellman-Ford and Floyd-Warshall algorithms are two different DP solutions to this problem. We can make use of this fact very well in our educational program, students who have very solid grounds in DP can discover Bellman-Ford and Floyd-Warshall themselves. To induce this, we can tell them to try finding the shortest paths with DP. If necessary, we can be more specific: in the case of Bellman-Ford: do a DP on the number of edges in the path. Floyd-Warshall is much trickier, the DP is done on the vertices inside the path. An excellent task that can help with this discovery is *CF - 295B. Greg and Graph*. In our third unit problem set, there is another task which is a great synthesis of DP, combinatorics, and directed acyclic graphs (DAG): *HR - Kingdom connectivity*, in which you have to find the number of different ways to go between two vertices of a directed graph.

4.4. Advanced problems

Since DP has so many applications, we keep on revisiting it with harder and harder tasks. It is not easy to categorize them, so we took two representative problems to symbolize these modules.

To solve the Longest Common Subsequence problem, we create a 2D array, where each cell corresponds to the subproblem taking the first `i` and `j` elements of the two sequences. For sequences `A` and `B`, the dynamic programming goes as follows:

```
DP[i,j] = max(DP[i-1,j], DP[i,j-1],                    (7)
              DP[i-1,j-1]+1 if A[i]=B[j])
DP[i,0] = 0, DP[0,j] = 0
```

We included various other exercises where we see this or a similar pattern. There is *M - Jelek (Signs)*, which is essentially the longest repeated substring problem that can be solved with DP in $O(N^2)$. In *M - Rúd felvágás (Stick cutting),* we are looking for the cheapest way of cutting up a stick to pieces. The solution to that problem is easier formulated with recursion, so it can be an example of memoization. *CF - 607B* is about palindromic substrings and it requires a very good understanding of the pattern where the subproblems are ranges in some sequence. Here we would like to point out that previously we mentioned a problem, *M - Fehér és fekete korongok, (White and Black Tokens*), which falls also into this category and we included it in GAME DP because there the subproblems are states of a game, so it is a good precursor to this module.

The Longest Increasing Subsequence (LIS) problem can be developed very well and there are numerous nice exercises on this topic. The $O(N^2)$ solution is a good first step, but here we want to focus on reaching the $O(N \cdot logN)$ solution combined with binary search. *M - Konténeroszlopok (Container Columns)* is a greedy task that can be viewed as the dual problem of it: divide a sequence into a minimal number of decreasing subsequences. In the greedy process, the ordered nature of subsequence endings can be observed, and the binary search is straightforward. What is not straightforward is that we calculate the length of the LIS too, which gives a lower bound for the result. This duality theorem also certifies that LIS is so beautiful that everyone should see. But we only uncover it after solving a task which can be reduced to finding the LIS: you have to build a tower using the maximum number of cubes, given some cubes with sizes and weights, and you can only place a smaller and lighter cube on top of another, *M - Kockákból legmagasabb torony (Highest Tower of Cubes)*. Sorting by one property is the first idea, after that it comes down to computing the LIS. There is a very hard task in this topic, *CF - 650D. Zip-line*, in which a firm understanding of the above is necessary, but no advanced data structures.

We call Bitmask DP the method when the subproblems we solve correspond to all subsets of a set. We usually represent a subset with a bit vector, that is usually stored in an integer. Transitions in this form of DP generally involve adding or removing one element of the subset, which can be done by bit manipulations, bitwise operators. An example problem would be *M - Vásárlások (Purchases)*, where we need to minimize the money spent on certain items, given their prices in different shops, with the constraint that we buy at most one item in each shop. The states of the DP, in this case, would be the `K` number of shops considered and the `S` subset of items bought so far. This type of DP can be connected to the inclusion-exclusion principle in some problems, for example *CF - 102128B. Cake Tasting.*

We applied DP with binary search, DP on ranges and DP on subsets with bit manipulations in the modules above, and there are numerous other scenarios that we haven't covered here. The topic of DP is very deep, there can be extremely difficult problems which are "only" DP. In our fifth unit, driven by data structures, we can see traces of DP in the construction step of many data structures (e.g. RMQ Sparse Table, Binary Indexed Tree, LPS table in KMP), so dynamic programming is a core concept throughout our curriculum. In this chapter, we gave an overview of it, showing the possibility of developing deep knowledge over a long period and many exercises.

5. Conclusion

In this paper, we described a possible curriculum for computer programming talent education in high schools. However, we consider it as work in progress, we will reflect on it and improve it based on experiences from putting it to practice. The main goal of the author's research is to create a system for computer science talent education similar to the existing one in mathematics organized by The Joy of Thinking foundation [7]. The curriculum was designed to respect the principles of Lajos Pósa's pedagogy in mathematics and is intended to form as a basis for our problem-based learning methodology in algorithmic programming.

Bibliography

1. J. Győri, P. Juhász: *An extra-curricular gifted support programme in Hungary for exceptional students in mathematics*. Teaching Gifted Learners in Stem Subjects. Routledge, London (2017) 89–106. DOI: 10.4324/9781315697147-7
2. S. Halim: Competitive Programming 3. Lulu Independent Publish (2013)
3. *Problemset, Codeforces*. (2020) https://codeforces.com/problemset.
4. W. Van Joolingen: *Cognitive tools for discovery learning*. International Journal of Artificial Intelligence in Education. Vol 10 (1999) 385–397.
5. A. F. Borthick, D. R. Jones: *The motivation for collaborative discovery learning online and its application in an information systems assurance course*. Issues in Accounting Education, Vol. 15(2) (2000) 181–210. DOI: 10.2308/iace.2000.15.2.181
6. S. L. Finkle, Ll. Torp: *Introductory Documents*. Illinois Math and Science Academy. (1995)
7. *The Joy of Thinking Foundation*, Hungary. http://agondolkodasorome.hu/.
8. J. A. Bibergall: *Learning by discovery: Its relation to science teaching*. Educational Review. Vol. 18(3) (1966) 222–231. DOI: 10.1080/0013191660180307
9. D. Katona, G. Szűcs: *Pósa-Method and Cubic-Geometry – A Sample of a Problem Thread for Discovery Learning of Mathematics*. Differences in Pedagogical Theory and Practice. (2017) DOI: 10.18427/iri-2017-0079
10. P. Juhász: *Hungary: Search for Mathematical Talent*. The De Morgan Journal. Vol 2(2) (2012) 47-52.
11. L. Pósa: *Matematika táboraim*. Természet Világa. Vol. 132, N°3. http://www.termeszetvilaga.hu/tv2001/tv0103/posa.html
12. M. Forišek: *IOI Syllabus*. (2018) https://ioinformatics.org/files/ioi-syllabus-2018.pdf
13. A. Laaksonen: Competitive Programmer's Handbook. (2018) https://cses.fi/book/book.pdf
14. *CP-Algorithms*. https://cp-algorithms.com/
15. *Geeks for Geeks*. https://www.geeksforgeeks.org/
16. A. M. Dehghan: *Algorithm Gym: Data Structures*. Codeforces blog. (2015) https://codeforces.com/blog/entry/15729
17. A. M. Dehghan *Algorithm Gym: Graph Algorithms*. Codeforces blog. (2015) https://codeforces.com/blog/entry/16221
18. S. Király: *How to teach computer programming if our goal is the International Olympiad in Informatics*. Teaching Mathematics and Computer Science. Vol. 9, no. 1 (2011) 13-25.

19. Á. Erdősné Németh: *From LOGO Till Olympiads - Talent Management In Grammar School.* Ph.D. Dissertation, Eötvös Loránd University, Doctoral School of Informatics, Budapest (2019) (in Hungarian). DOI: 10.15476/ELTE.2019.007

20. Á. Erdősné Németh, L. Zsakó: *Grading Systems for Algorithmic Contests.* Olympiads in Informatics. Vol. 12 (2018) 159–166. DOI: 10.15388/ioi.2018.13

21. Á. Erdősné Németh, L. Zsakó: *The Place of the Dynamic Programming Concept in the Progression of Contestants' Thinking.* Olympiads in Informatics. Vol. 10 (2016) 61–72. DOI: 10.15388/ioi.2016.04

22. M. Forišek: *Towards a better way to teach dynamic programming.* Olympiads in Informatics. Vol. 9 (2015) 45–55. DOI: 10.15388/ioi.2015.05

23. P. Szlávi, L. Zsakó: *Methodical programming: Programming Theorems*. Eötvös Loránd University, Faculty of Science, Department group of Informatics, Budapest (1996) (in Hungarian).

24. *HackerRank*. https://www.hackerrank.com/

25. *CodeChef*. https://www.codechef.com/

26. *CS Academy*. https://csacademy.com/

27. *Sphere Online Judge*. https://www.spoj.com/

28. *UVa Online Judge*. https://uva.onlinejudge.org/

29. *PEG Online Judge*. https://wcipeg.com/

30. *Mester*. http://mester.inf.elte.hu/

31. *ICPC Live Archive*. https://icpcarchive.ecs.baylor.edu/

32. *Timus Online Judge*. http://acm.timus.ru/

Authors

NIKHÁZY László

Eötvös Loránd University, Faculty of Informatics, Department of Media and Educational Informatics, Budapest, Hungary, e-mail: laszlo.nikhazy@gmail.com

About this document

Published in:

CENTRAL-EUROPEAN JOURNAL OF NEW TECHNOLOGIES IN RESEARCH, EDUCATION AND PRACTICE

Volume 2, Number 1. 2020.

ISSN: 2676-9425 (online)

DOI:

10.36427/CEJNTREP.2.1.399

License