Schema-Based Instruction with Enumerative Combinatorics and Recursion to Develop Computer Engineering Students' Problem-Solving Skills

International Journal of Engineering Education Vol. 36, No. 5, pp. 1505–1528, 2020
Printed in Great Britain

0949-149X/91 $3.00+0.00

Schema-Based Instruction with Enumerative Combinatorics and Recursion to Develop Computer Engineering Students' Problem-Solving Skills*

GIANPIERO CABODI, PAOLO CAMURATI, PAOLO PASINI, DENIS PATTI and DANILO VENDRAMINETTO
Politecnico di Torino, Dipartimento di Automatica e Informatica, c.so Duca degli Abruzzi 24, I-10129 Turin, Italy.
E-mail: {gianpiero.cabodi, paolo.camurati, paolo.pasini, denis.patti, danilo.vendraminetto}@polito.it

Learning and teaching problem solving is a hard task, no matter the domain. Computer Science is no exception. Recursion is a paradigm often used for problem solving, but it is non-intuitive and it is unnatural. Most second programming courses (*CS2-level*) for Computer Engineering students apply recursion to mathematical problems or basic recursive data structures with a limited focus on problem solving. Third programming courses (*CS3-level*) deal with search and optimization problems and they use recursion because of its ability, due to its backtracking mechanism, to explore the whole solution space. However, most of them do not rely on a systematic and well-formed approach to teaching this approach to problem solving. Our main contribution is to adopt schema-based instruction for recursion-based problem solving, where schemas come from Enumerative Combinatorics. This is the core of our attempt at developing second-year computer Engineering students' problem-solving skills. We provide the students with these schemas as templates in the C language to guide them step-by-step in solving search and optimization problems with uninformed and complete algorithms. To extend the applicability of this approach to other than small-size problems, we show students how they can introduce pruning to limit search while keeping it complete. We present experimental evidence we gathered for our second-year *CS2+CS3* programming course for Computer Engineering students at Politecnico di Torino, a major technical university in Italy. We evaluate students' perception of the approach in terms of understanding and of ability to apply it. We compare students' perception to faculty expectations and we evaluate students' performance in terms of improvement in the success rate at exams. Data prove that the approach we adopt is beneficial both in terms of quantitative results (success rate at exams) and qualitative results (knowledge and skills acquired by students).

Keywords: applied computing; education; schema-based instruction; theory of computation; algorithm design techniques; recursion; problem solving

1. Introduction

It is hard to say whether there are more differences or similarities between Computer Engineering and Computer Science. They originate from Electrical Engineering and from Applied Mathematics respectively, the former has always focused more on developing skills in designing software, hardware and systems, the latter on theoretical foundations of computation. The former is historically more hardware-oriented, the latter more software-oriented. In practice, the overlap between these two fields of studies is large, as they both have as a goal computers, their study and their use. Traditionally Computer Science curricula have focused more than Computer Engineering ones on programming, algorithms and data structures. Because of the convergence of the two fields, this is not true anymore, as programming has become a must also for Computer Engineering students.

Programming is a cognitive activity that, while solving a problem, involves high-order aspects like mathematical skills, abstraction abilities and critical thinking [1, 2]. Because of its nature, it makes decontextualized teaching and mere rote learning [3, 4], still common approaches to teaching and learning in Engineering, totally useless.

In general, teaching programming to undergraduate Computer Engineering students follows two main approaches:

1. Select a language and then teach the syntax and the main constructs of that language, with problems mostly exploited as examples of use of the constructs.
2. Teach the students how to solve problems and resort to programming as a tool to automate the process.

Language-oriented programming courses often focus more on the technicalities of the language, rather than on its use. As a result students master a large number of details, but often they fail to use them when facing a problem to solve, even when it's not a real-world one. They know the syntax, they understand the semantics of each construct, but they are unable to write code solving a specific problem.

* Accepted 28 February 2020.

This is true in particular for novice programmers, but not restricted to them [5]. According to [6] this is due to the lack of design strategies and problem-solving skills. Problem-solving oriented approaches consider the language as a tool and not as a goal, thus they aim at identifying strategies and at designing algorithms and then, last but not least, at implementing them as code written in a specific programming language.

In our experience the second approach is much more challenging for the students but also much harder to learn, as problem solving is a typically creative activity, that relies on knowledge, intuition and intelligence. However, being creative doesn't mean that everything must start from scratch, as there is a vast background of pre-existing knowledge developed over the years that comes to the aid. In this sense our approach is a mix of deductive and inductive teaching [4]: it is deductive because we leverage on knowledge developed in Mathematics (if one knows what a permutation is, there is no need to examine several examples to understand what an anagram is), it is inductive because the connection to problems coming from the real world is stressed right from the beginning (e.g., problems coming from programming contests or job interviews), thus motivating the students. It is hybrid because it cross-fertilizes Enumerative Combinatorics and problem solving in Informatics, bridging a gap that purely deductive approaches don't cross.

Our approach belongs to the broad category known as **schema-based instruction** [7], originally developed to help students who had difficulties in learning Maths. An interesting experience about Geometry problem solving is reported in [8]. According to [7], a "*schema is a vehicle of memory, allowing organization of an individual's similar experiences in such a way that the individual:*

- *can easily recognize additional experiences that are also similar, discriminating between these and ones that are dissimilar (identification knowledge);*
- *can access a generic framework that contains the essential elements of all of these similar experiences, including verbal and nonverbal components (elaboration knowledge);*
- *can draw inferences, make estimates, create goals, and develop plans using the framework (planning knowledge);*
- *can utilize skills, procedures, or rules as needed when faced with a problem for which this particular framework is relevant (execution knowledge)*".

The purpose of identification knowledge is to recognize that a schema might fit the problem. Conversely, elaboration knowledge adapts the schema to the problem, resulting in a mental model. Planning knowledge exploits the schema to define the set of actions needed to solve a problem. Execution knowledge allows the individual to carry out the planned steps.

Schema-based instruction [7]:

- "*unifies declarative and procedural knowledge within a broader framework;*
- *de-emphasizes the quantity of factual bits of information that the student acquires, as its goal is to cultivate learners who will be active problem-solvers in the field – not to produce students who have a large store of passive or inert knowledge;*
- *introduces the domain to students in a top-down rather than a bottom-up way*".

We tailor schema-based instruction to our specific domain: teaching recursion-based problem solving to Computer Engineering students. Why recursion? Among the approaches to problem solving in Informatics, recursion-based techniques play a key role in educational terms: recursion is a non intuitive concept and a powerful but unnatural divide-and-conquer paradigm, difficult to teach and to learn [9–11]. Many second programming courses (*CS2-level*) deal mainly with the concept of recursion and illustrate it in terms of mathematical recursion, where there is an immediate translation from a recursive formulation of the problem into code. Moreover, in general, there is no need to make choices, as every step follows immediately from the previous ones. Most courses apply recursion to classical recursive objects, like lists, trees and graphs, introducing functions on Abstract Data Types or algorithms that are well known from the literature. Some courses resort to visualization [12] or to video games [13] to teach novices recursion. Fewer courses, mainly at *CS3-level*, use it for more advanced problem solving based on uninformed exhaustive search in a solution space generated by taking into account all possible choices. Their key to success is the intrinsic backtrack mechanism that guarantees completeness. Their limit is the size of the solution space that precludes applicability beyond medium-small problem instances. Pruning the search space or introducing heuristics to bias choices is normally left to more advanced courses. To the best of our knowledge no course is fully based on schema-based instruction, very few introduce some form of schema as combinatorics-based solution space modelling. Our schemas originate from Enumerative Combinatorics and span the entire portfolio of models. We cultivate second-year undergraduate students in Computer Engineering attending a *CS2+CS3-level* course top-down and step-by-step: Enumerative Combinatorics models represent identification knowledge, their accommodation to the problem at hand

relates to elaboration knowledge, algorithm design is planning knowledge and C code is a result of execution knowledge. Advanced techniques for searching the solution space fall into the sphere of AI or Operational Research, that are beyond the scope of this paper. We do not aim at skills that boost performance, needed for example for programming contests: our main concern is that the students become able to master complete recursive solutions, with the awareness of their intrinsic complexity, that they are able to tackle just by a proper selection of the search space, and by limited and straighforward pruning techniques, plus a selective application of dynamic programming and memoization.

In the context of a second-year Computer Engineering *CS2+CS3* course, focusing on how to effectively teach problem solving, leveraging recursive techniques, this paper addresses the following research questions:

1. What challenges do students encounter in learning how to solve problems with recursion?
2. What challenges do lecturers encounter in teaching problem solving based on a recursive paradigm?
3. Is schema-based instruction a suitable approach to improve second-year students' problem-solving skills?

2. Background

This section describes the context of our experience, the main issues lecturers have to deal with when teaching recursion and problem solving, and the contributions and organization of this paper.

2.1 The Context

Algoritmi e Programmazione (Algorithms and Programming) is a second-year undergraduate course mandatory for Computing Engineering students at Politecnico di Torino (around 450 new students each year). It follows a first-year course (*Informatica, CS1-level*) where students acquire basic programming skills in the C language (syntax, datatypes, input/output, loops and conditionals, arrays and multi-dimensional arrays, C structures as composite data types, functions and files). First-year problem solving is restricted to simple cases. Elementary notions on the representation of numerical data, Boolean algebra and computer architecture are also part of the syllabus. The second-year course, at *CS2+CS3-level*, builds on the first year:

- More advanced C language features, including pointers and dynamic memory allocation.
- Abstract Data Types implemented in C (trees, graphs, symbol tables, etc.) and related algorithms.
- Analysis of algorithmic complexity.
- Problem-solving paradigms (recursive divide-and-conquer, dynamic programming, greedy algorithms, etc.).

The expected learning outcomes are the ability to solve with complete algorithms medium-small problems, analyzing their complexity, to design and use classical Abstract Data Types and algorithms, and to master the C language as a tool to support problem solving. The course spans one semester (14 weeks), it is worth 12 ECTS credits, it consists of 38h of lectures concerning complexity, algorithms, and problem-solving paradigms, 38h of lectures concerning advanced C features and their use in algorithms, data structures, and problem solving, 20h of in-class problem-solving sessions, and 24h of assisted in-lab training. As 1 ECTS credit is worth 25h of student work, students are supposed to devote additional 180h of personal work to this course, including 70h for unassisted training and lab assignments. Available material includes a textbook in Italian [14], specifically covering recursion, video-recorded lectures, transparencies, assignments and commented solutions to assignments. Access the course website is restricted to enrolled students.

Teaching resources are constrained (2 professors, 5 assistant professors, a varying number of undergraduate and graduate students). Until academic year 2017/18 the same teaching resources served 2 parallel classes. Starting from academic year 2018/19 a third class in parallel has been offered, reducing class size from about 220 to about 150 and lab classes size from about 110 to about 75. Class size reduction is one among the course setups considered in [15]. Its effect is analyzed in Section 5. Student population and availability of teaching resources greatly limit the possibility of experimenting with new teaching approaches, like the ones reported in [3]. Therefore the teaching-learning pattern relies on lectures (participation and control of the educator) and labs where educator and students share participation and control [3]. The paradigm for in-lab training is constructivism, especially as regards to the internalization of the recursive models and to the application of individual knowledge to problem solving. Lectures follow mainly the deductive teaching paradigm, but they include application of theoretical concepts not just to examples, but to problems. During in-class problem-solving sessions, the lecturer follows a *think-aloud* protocol [16], verbalizing his thought process while analyzing a significant problem, discussing the strategies to solve it and then defining a high-level structure of

the code that solves it. Reported experience with think-alouds in programming courses confirms that it is a very good approach, appreciated by students and relatively lightweight for faculty [17]. Assignments are on a weekly basis, students may work individually or in a group, they receive counselling during in-lab sessions or by e-mail, but most of the work is done at home. Exercises are classified as basic, intermediate and advanced. They may be mandatory or optional. The goal of basic exercises is to make the foundations of a given topic more solid. They are optional: if the student feels confident enough, he/she doesn't need to hand them in for evaluation. Intermediate exercises are more complicated, require selecting a model from Enumerative Combinatorics or to define a divide-and-conquer strategy. They are mandatory, as this is the level students are supposed to reach. Advanced exercises may require combining several models, may have more difficult solution validation functions, solution space pruning may be requested. They are optional, most top students solve them, as they find them challenging. As an example, a basic exercise is to write the code for the greatest common divisor of 2 positive integers given a recursive formula. An intermediate one is to compute a minimum vertex cover, given the list of nodes and edges of an undirected graph. An advanced exercise may be a tile game, where the goal is to arrange tiles on a board according to given rules, maximizing the score. Assignments are handed in for evaluation in 4 groups of 3 assignments each, so students have around 3 weeks to complete each group. This allocates enough time also for slow programmers. Faculty verify selected assignments with a personal interview. The goal of the interview is not only to rank the assignment, but also to understand whether the student has built his/her own mental model of the problem and has defined a suitable strategy for solving it. This interactive phase is a form of feedback for the faculty, both as regards to the individual students (e.g., fixing wrong models) and to the course as a whole (e.g., clarifying aspects still perceived as obscure).

The exam consists of both a written and of an oral part. There are two written examination modes:

- Problem solving at large, i.e., writing a program in the C language to solve a problem where the candidate is totally free to make his/her choices. This entails identifying the underlying model, select the data structures, write a function to verify that a solution is valid, write an objective function for optimization problems, identify possible pruning techniques.
- Constrained problem solving, where there is a guidance in the selection of the model and ranking is about the ability to implement it as a C language program, rather than to discover it.

The first mode leads to full marks, the second one has an upper bound on the marks. The oral exam has two goals: to assess knowledge on algorithms and data structures and to test the ability to write C code on the fly. As regards to the latter issue, the exam looks like a job interview: given a well-defined problem, show the problem-solving skills the candidate has acquired and the ability to implement the solution as a C program. Duration is around 30 minutes. Marks are relevant in the Italian University system, as they are the basis for rankings related to academic and economic benefits and to employment opportunities.

2.2 Recursion-Based Problem Solving

Among the approaches to problem solving in Informatics, recursion-based techniques play a key role in educational terms: recursion is a non intuitive concept and a powerful but unnatural divide-and-conquer paradigm, difficult to teach and to learn [9–11]. The next sections analyze the issues in learning and teaching recursion and problem solving. We conclude that we modify the way we teach, by changing the contents we teach, according to the schema-based paradigm. Our schemas originate from cross-fertilizing Enumerative Combinatorics and recursion-based paradigms. The goal is to equip students with a problem-solving methodology and portfolio of models applicable to a large set of cases.

2.2.1 Issues in Learning (and Teaching!) Recursion

Students feel recursion is "*obscure, difficult and mystical*" [18] when they start approaching it and conversely for lecturers it is "*one of the most universally difficult topics to teach*" [19, 20]. It thus challenges both categories. Many academics have studied the multiple facets of teaching recursion. Excellent surveys are [21] and [22]. According to [22] students progress through 3 steps:

1. *Comprehension*: they understand what recursion means and what it is intended to do, building their own mental model [23]. From the lecturers' point of view, the main point is to orient them to develop a correct model. A viable one is the "copies" model [24, 23], where recursive functions generate new instantiations of themselves, passing control and possibly data forward to successive instantiations and back from terminated ones. Lecturer guidance is needed: in fact it often happens that, without guidance, the spontaneous model students end up with is not viable or it

is incomplete, like the ''looping'', the ''active flow'', and the ''syntactic'' ones [25]. The first one considers recursion as a single object with an entry point and an iterative part, the second one correctly reaches the base case but fails to understand how recursion unrolls itself, the third one is limited to a mere template ''base case and recursive case''.

2. *Evaluation*: given a recursive function written in the C language, they learn how to simulate its behaviour on simple datasets (dry-run).
3. *Construction*: they first interpret the simple recursive examples they have understood and traced at the previous steps as instances of a general-purpose divide/decrease-and-conquer problem-solving approach. They then apply an ''analysis and synthesis'' method [23] to start solving problems on their own, breaking a problem into smaller and smaller subproblems, until they find trivially solvable ones, building the solution to bigger problems thanks to the solutions to smaller ones, encoding their approach in the C language and running it on significant examples. Mathematical functions (factorial, Fibonacci, Bell and Hofstader numbers (more in [26]), greatest common divisor, etc.) expressed as embedded recursive functions [27] offer a large set of examples to start with, as well as recreational Mathematics (Towers of Hanoi, Sierpinski's triangle, more in [28]) and basic Informatics (recursive list processing, trees, etc.). These targeted practice exercises, possibly supported by tutoring software [29], are considered the best way to learn. Problem-solving skills at this step are still limited, but students become aware that divide/decrease-and-conquer strategies are powerful paradigms. Most of them succeed in solving such problems because they are familiar, if not with the problems themselves, at least with the domain (Mathematics) and they are good at applying templates to situations that are already known [30].

A search in the web for open-access teaching material has shown that almost all *CS2+CS3-level* courses in major Universities around the world follow this approach with more or less emphasis on topics like the model of recursion and its implementation on a computer, dry-runs and the ability to understand and possibly design recursive solutions to simple problems. Most courses resort to the deductive approach.

2.2.2 Issues in Learning (and Teaching!) Problem Solving

Problem solving is an activity that encompasses several domains, ranging from philosophy to science, engineering, psychology and computer science. Teaching problem-solving skills has always been recognized as a major issue in education starting from elementary school to University and life-long learning [31, 32]. The goal of *problem solving* in *Informatics* is to let students develop a mindset and an approach to systematically analyze problems, to design strategies that, starting from input data, generate output data, i.e., results, possibly compliant with external constraints, and eventually to automate the process by means of code written in a programming language.

A non-exhaustive classification of common problems is:

- *Pure-computation problems:* for which in general there exists a mathematical formula that we almost immediately transform into code. Such problems are rather straightforward, as there is no need to make a choice selected from a pool of available ones. They are more akin to exercises than to problems, the difference being that problem solving needs creativity, whereas exercise solving just needs the application of known procedures. Examples of this class are mainly mathematical problems, including the ones suitable for a recursive solution listed in Section 2.1.1.
- *Decision problems*: for each problem instance the solution is either yes or no. Among them a vast class is represented by *verification problems*: given a problem instance and a possible solution, check whether it is valid or not.
- *Search problems*: for each problem instance a solution, i.e., a string of bits that encodes an information item, is an element in the *solution space*. In fact not all solutions are valid, only the ones that comply with given constraints. One valid solution, if it exists, may be enough. If all valid solutions are sought after, it is an *enumeration problem*. Examples are the 8-queens problem, Sudoku, listing all simple paths from a source node in a graph, etc.
- *Optimization problems* where all solutions in the solution space are valid, but the result is the one or the ones that minimize or maximize an objective function, e.g., loss, cost, utility, fitness, etc. Enumeration is required if seeking an absolute maximum or minimum. Many examples come from graph theory, like single-source longest paths, maximum independent set, the travelling salesman problem, etc. Search and optimization problems turn into decision problems when there is a bound on the admissible value of the solution and the question becomes whether such a solution exists or not.

Other classes include problems solved by means of numerical methods or simulation, that are beyond the scope of *CS2+CS3-level* courses.

Teaching problem solving even in a restricted domain like Informatics is a hard task, as "*it is not science, but part art and part skill*" [33]. Just looking at the syllabus of the average programming course, it is rare not to find the claim that that course teaches problem solving. However, as observed by [34] and [1], in most cases problem solving remains nebulous and scattered in a bunch of programming knowledge. Widely used approaches when the target is a *CS3-level* course for Computer Engineering students are:

1. Teach the tool (the programming language), then face the students with problems and let them solve them without guidance.
2. Teach them the programming language, a large set of knowledge about data structures and algorithms known from the literature, then face the students with problems and let them solve them on their own.
3. Add to the previous step examples of solved problems. The students proceed by imitation and improvement ("imitatio et aemulatio" according to the letter "De imitatione" written in 1513 by the Italian Renaissance scholar Pietro Bembo in a totally different context).
4. Add to the previous steps some problem-solving paradigms, like divide/decrease-and-conquer, dynamic programming, and greedy approaches. Heuristics-based paradigms, simulated annealing, integer linear programming and the use of solvers are beyond the scope of such a course.
5. Add to the previous steps or partially replace them with inductive-based learning by experience on real problems and/or assignments, under cooperative (student teams) and faculty-supervised working models.

The first approach has limited educational value and works just on simple problems. Moreover, it doesn't take into account the existence of a vast amount of knowledge. The second one keeps knowledge and practice apart, putting the burden of bridging them on the students' shoulders. Both are common in deductive teaching approaches. The effectiveness of the third one depends on a careful choice of representative examples and on the availability of a repository of classified problems like [33]. It is a form of *learning by analogy*, a type of knowledge transfer where the problem is first represented, then an analogous one is searched for and its solution adapted [35]. Resorting to paradigms is a important step in helping the students create a problem-solving mindset. Paradigms introduce a methodology, which is general-purpose enough not to reduce problem solving to exercise solving, but still requires creativity to find a solution. The 3rd, the 4th and the 5th approaches mix to different degrees deduction, i.e., methodologies and paradigms, and induction, i.e., experiential learning. Lecturer guidance may intervene at any phase. Our appoach belongs to the 4th group, but it is not an analogy-based reasoning: we do not look for analogous problems, we rather identify a schema, i.e., a mathematical model, of the problem based on Enumerative Combinatorics. Code for general-purpose models is provided, independent of specific data, as inputs and outputs are encoded as integers. This code is then adapted to the inputs and outputs of the current problem and extended with solution validation functions, optimality functions and pruning. Our approach best fits the broad class of schema-based instruction, as discussed in Section 1.

A search in the web for open-access teaching material has shown that the 2nd and 3rd approaches are the most common ones, but the paradigm-based approach is becoming increasingly popular. Nevertheless, as regards to recursion-based paradigms for search and optimization problems, it is uncommon to find modelling of the solution space and templates for its exploration in a schema-based instruction context.

2.3 Contributions

Given the constraints reported in Section 2.1, namely a large student population and limited teaching resources, experimenting with new teaching approaches, like the ones reported in [3] is not feasible in our context. Nevertheless some of the activities surveyed in [15] are applicable to our case, namely:

- Peer support: support by hired peer mentors, in general past or graduate students, during lab sessions.
- Support activities, e.g., extra hours where faculty tutor students, additional support channels, like e-mail or a faculty-moderated forum on the course site.
- Contents change.

The novelty of our approach resides mainly in the change in the contents in agreement with schema-based instruction, a method of teaching problem solving that emphasizes both the semantic structure of the problem and its mathematical structure. We have modified what we teach by cross-fertilizing Enumerative Combinatorics and problem solving, importing the models from the former to typify the latter, guiding students with a template-based approach to solve search and optimization problems with uninformed and complete approaches.

Algorithms are *uninformed*, i.e., they have no specific knowledge of the problem at hand and *complete*, i.e., they are capable of exploring the whole solution space. Models also serve the purpose of classifying problems that would otherwise be presented in an unstructured, empirical way.

2.4 Schema-Based Instruction for Teaching Problem Solving

We focus on search and optimization problems that require exploring a solution space, because these are inherently suitable for a recursive approach with backtracking. We describe the solution space in terms of models coming from Combinatorics. Enumerative Combinatorics [36, 37] courses are common in undergraduate curricula in Mathematics, but, due to their context, they do not focus on problem solving. Basic Combinatorics courses are found in Computer Engineering undergraduate curricula, but their aim is to count the elements that belong to the solution space, rather than to list them. Counting is in fact the basis for Probability Theory. We provide the students with abstract schemas implemented as a portfolio of templates in the C language that they customize to the specific case they are dealing with. The steps they follow when solving a problem are clear and predefined and follow the three stages of computational thinking presented in [38]:

1. Problem formulation:
   - *Understanding the specifications* is, in our experience, by no means a trivial task for the average student. Many students are increasingly limited in the use of natural language and in the amount of information they exchange. They often find it difficult to understand texts that are long and complex not per se, but in relation to their scale. Paying attention to writing the specifications using plain, yet not poor, language is a must. Resorting to mathematical formulas is feasible, it has a significant advantage in terms of conciseness and precision, yet some students find it hard to understand them. This is probably due to a difficulty in abstracting concepts and in building mental models for what they read. For this reason examples are of the uttermost importance and we must select them with care to be representative of the problem and not just of some corner cases.
   - *Identifying the Enumerative Combinatorics model that describes the solution space* (identification knowledge in the schema-based instruction approach): first students have to identify which are the choices (groups in Combinatorics terms), then to check whether order matters, whether elements are unique and whether they may appear in several instances. If there is more than one model, they should select the one that best fits the problem. This is the most difficult step, as it requires abstraction capabilities.
2. Solution expression (elaboration, planning and execution knowledge in the schema-based instruction approach):
   - *Instantiating a template* for the model selected at the previous step as a skeleton of the solution; the template provides a base case and a scheme for recursive calls.
   - *Representing the solution*, usually as a single set (or two sets, current and best solution, in optimization problems), whereas we typically avoid a data structure to collect the full enumeration of possible solutions. If needed, we usually just ask to print them.
   - Filling the *base* case, which typically means setting up:
     – A function to *check* the solution.
     – Handling solution *scoring* and/or *weighting*, in case of optimization problems.
3. Solution execution and evaluation:
   - The solution is validated on small datasets.
   - *Improving the solution*: the size of the solution space grows exponentially and, in order to make the code developed at the previous step applicable to at least medium-size problems, it is necessary to resort to pruning techniques. A simple way of pruning the space is to subordinate recursive calls to validity checks, i.e., imposing constraints as soon as possible and not just in the base case. Another aspect that allows reducing space size without loosing completeness is identifying symmetric solutions. Exploiting symmetry is beyond our scope, as we believe it is too advanced for the students' level. In optimization problems, *branch-and-bound* strategies are also considered, though they are not a major focus for us.

Courses that adopt [33, 18] and similar books as textbooks follow the same approach as ours, but to a more limited extent, both because they are neither conscient nor rigorous in applying schema-based instruction and because they use only a subset of schemas based on Enumerative Combinatorics. We systematically introduce:

- A unifying framework for recursive objects like sets and multisets in Combinatorics.
- The 2 counting principles: the *principle of addition* (a.k.a. rule of sum) and the *principle of multiplication* (a.k.a. rule of product).

- The 6 models: *simple arrangements*, *arrangements with repetitions*, *simple permutations*, *permutations with repetitions*, *simple combinations*, *combinations with repetitions.*
- The *powerset of a set* when the solution space is the set of all the subsets of that set.
- The *partitions of a set* when the solution space is the set of non-empty subsets such that each element belongs to one and only to one subset.

2.5 Outline

Section 3 introduces the background notions on Enumerative Combinatorics. Section 4 describes templates for them written in the C language, including the use of pruning techniques to improve scalability to medium-size cases and a discussion of significant examples students are requested to solve as assignments or at exams. Section 5 analyzes data collected over a period of 7 years to evaluate the impact of this approach on the students' problem-solving skills. Finally, Section 6 concludes with some summarizing remarks and points to future work.

3. Modelling the Solution Space by Means of Enumerative Combinatorics

Let us consider the problem of selecting a subset of elements from a given set or multiset. We just consider finite sets $C = \{c_0, c_1, \ldots, c_{n-1}\}$ represented by the explicit collection of their elements. A multiset extends the concept of set by taking into account distinguishability of elements: a multiset is a set with repeated identical, thus undistinguishable, elements. A partition of a set captures the idea of grouping all set elements into non empty and disjoint subsets. We just consider multisets with a finite base set, represented by the explicit set of base elements, each one labelled by its multiplicity. The multiset $C = \{m_0 \cdot c_0, m_1 \cdot c_1, \ldots, m_{n-1} \cdot c_{n-1}\}$ is thus a set where c_i element is repeated m_i times. A given multiplicity can be either finite or infinite: we consider both cases, as they support explicit representation. A partitioned set is represented by an explicit enumeration of subsets, or by the original set, with a proper element-partition labelling.

When solving non-trivial problems, consisting in selecting choices from a given set or multiset C of available ones, the solution $S \subseteq C$ might be either e set or a multiset. Partitioning problems either operate on possibly already partitioned sets or multisets, or deal with the problem of finding a partition of a given set or multiset.

Depending on the problem, we might be interested in an ordered or unordered set or multiset. Order is usually considered only when generating a solution S, not when providing the choice set C: two solutions S_1, S_2 might differ either for some elements or just in terms of their order. Order in multisets does not distinguish among multiple instances of the same element. In the most general case, the space of all solutions is thus represented by the powerset of set or multiset C when order doesn't matter, or by all permutations of elements of the powerset of C when order matters.

As an example, let us consider the problem of generating the anagrams of a given word or the equivalent problem of generating all numbers obtained by different orderings of a given set of digits: the problems are from sets to ordered sets, when no duplicates are considered, whereas they are from multisets to ordered multisets if duplicates are allowed. Generating numbers from a given set of digits and/or strings from a set of allowed characters, are problems from sets to ordered multisets. Here order matters, as the output of the problem is a string or a number in a positional representation. Other problems do not care about the order, e.g., finding the set of the 4 most frequent characters in a given text: this is a problem from a multiset[1] to a set.

Whenever a given solution S is a subset of C, it can be provided either as an ordered list or as an array, i.e., a tuple where the order matters or not, or as a proper binary labelling for sets or integer labelling for multisets and/or ordered sets, of the elements of C. With partitioning problems, we extend the above representations to multiple lists or arrays or integer labellings, where each element of C is labelled with the index of a partition.

The problem-solving strategy we propose is based on an explicit enumeration of the solution space. We enumerate by recursively and incrementally taking or collecting possible decisions or choices, following two alternatives:

- At each decision or choice step, we select one among the at most n available elements of C. This element is the one we add to the solution, where the solution is an ordered or unordered set or multiset, depending on the problem.
- At each decision or choice step we consider one of the n elements of C by properly labelling it. The final solution is given by the labelling. A subset is identified by a $0/1$ labelling, an ordered subset by a natural number labelling.

Our uninformed and complete problem-solving technique resorts to an incremental approach to

[1] The starting set is actually a multiset, when counters of occurrences for single characters are already given, it is the base set if we start from the text and still have to count repetitions.

explore the solution space: starting from the empty solution $S = \emptyset$, step-by-step we:

- Extend by one (s_i) the partial solution selecting one of the $c_j \in C$ choices. We terminate when we have made all decisions, e.g., when the cardinality of the solution is k (whenever given).

or

- Decide whether to select or how to label the current $c_i \in C$ choice. We terminate when we have labelled all choices. The final solution is given by the labelling.

We deem explicit enumeration more understandable for students than implicit enumeration techniques or constructive, recursive divide-and-conquer approaches where the solution is built by combining simpler ones. The first approach, the one that at each step extends the solution, is more intuitive and thus more easily understood by students for most cases where they know in advance the cardinality of the solution and therefore when they terminate they don't have to perform additional checks.

In this case a *search tree* represents the solution-building process: the tree:

- has height k, the cardinality of the solution, i.e., the number of decisions we have to make;
- has degree n, the maximum number of possible choices;
- the root is the initially empty solution $S = \emptyset$;
- choices selected so far and representing partial solutions label internal nodes;
- leaves are base cases (solutions) and they form the solution space.

We may check solution validity on a leaf or earlier ("pruning" the space) to anticipate constraints. We check solution optimality on a leaf or we early discard that solution in a branch-and-bound approach [7], not considered in this course. The following pseudo-code illustrates an algorithm based on a generic data structure DS to explore a solution space:

```
1. Search():
2.   insert initial item in DS
3.   while DS not empty
4.     extract item from DS
5.     if solution
6.       return success
7.     for all valid choices
8.       apply choice
9.       put resulting items in DS
10.    return failure
```

Algorithm 1. Solution space search based on a generic data structure DS

If DS is:

- a *queue:* search proceeds *breadth-first*;
- a *stack:* search proceeds *depth-first*;
- a *priority queue:* search proceeds *best-first*.

Our approach to exploring the solution space is *depth-first*, *complete*, *uninformed* and based on *recursion*. We have n choices, where n is the cardinality of set C, and we collect decisions about them in groups of cardinality k according to specific rules. Combinatorics [39] is the branch of Discrete Mathematics concerned with counting how many finite elements, choices in our problem-solving context, satisfy certain properties. There are two basic principles: addition and multiplication. They apply whenever there are several sets of choices. When there is a single set of choices, the principle of multiplication gives rise to more specific models that group elements together according to 2 criteria [1]:

- Uniqueness of the elements: if elements are unique, we are dealing with sets, otherwise with multisets.
- The importance of ordering: 2 groups with same elements but different orderings are the same or not.

This results in a portfolio of schemas that we exploit for solution space exploration, listed in Table 1 where rows are labelled in terms of how the C inputs and choices are grouped as either sets or multisets and columns in terms of how the S outputs or decisions are grouped (sets, multisets, tuples, tuples with repetition). Element multiplicity is known a priori for *permutations with repetitions*, thus the input is a finite multiset. It is typically free, though with an upper bound k, for *arrangements with repetitions* and *combinations with repetitions*, so in this case the inputs are infinite multisets[2]. In terms of data structures, both sets and tuples are represented as arrays in the C language, where a predecessor/successor relation is implicit. It is up to the context to interpret whether this relation matters, like for tuples or not, like for sets. This remark allows us to consider *simple combinations* as a particular case of *simple arrangements* when just one of the possible orderings is considered.

We first introduce the two principles of addition and multiplication, that solve the problem of selecting single elements out of sets in a group of sets or, equivalently, in a partitioned set. We then describe combinatorics problems on sets and multisets, and finally face the problem of partitioning a given set.

[2] As already noted, the base set is finite, whereas we consider repetitions as free, i. e. infinite.

Table 1. Inputs, outputs, and models

	Output			
Input	**Set**	**Multiset**	**Tuple**	**Tuple w/reps.**
finite set	simple comb.		simple arrang.	
finite set			simple perm.	
finite multiset		comb. w/reps.		arrang. w/reps.
infinite multiset				perm. w/reps.

The *principle of addition* applies to b sets $B_0, \ldots, B_{b-1}$. If such sets are pairwise disjoint they are a partition of a global set C of choices. Let $n_i = |B_i|$ represent the size of the ith set. If we can pick just one element from a set, then selecting one element ($k = 1$) from any of the b sets can be done in $\Sigma_{i=0}^{b-1} n_i$ ways. When sets are not pairwise disjoint, this principle evolves into the *principle of inclusion/exclusion*.

The *principle of multiplication* states that if k elements $x_0 \ldots x_{k-1}$ are selected in sequence each from a set of cardinality $n_0 \ldots n_{k-1}$, the k-tuple of elements $(x_0 \ldots x_{k-1})$ is built in $\Pi_{i=0}^{k-1} n_i$ ways.

We use the principle of addition for very few non-trivial problems. Application examples of the principle of multiplication are much more common. Though the principle applies to both overlapping and non overlapping sets, we found that for students the problems of the latter category are generally easier to understand and to solve. We thus explicitly use the principle of multiplication whenever there are several disjoint sets of choices, forming a partition of a set C, and we have to build the solution picking one choice from each subset. Considering overlapping sets, typically generated by the given choice set, taking into account order and repetitions, the application of the multiplication principle originates the models described next.

A *simple arrangement* $A_{n,k}$ of n distinct elements of class k is a tuple composed of k out of the n elements ($0 \leq k \leq n$). As elements are distinct, the input is a set C. As the output is a tuple, order matters and the adjective simple implies that in each tuple there are exactly k non repeated elements. Two arrangements differ if there is at least a different element or if the same elements appear in different order.

An *arrangement with repetitions* $A'_{n,k}$ of n distinct elements of class k is a tuple composed of k out of the n elements ($0 \leq k$) each appearing at most k times. As in the case of simple arrangements, order matters. Unlike simple arrangements, elements may appear several times. The input set is actually an infinite multiset, having C as base set, and each element appearing with infinite multiplicity. Two arrangements with repetitions differ, in addition to what holds for simple arrangements, also in the case the elements are the same, but appear a different number of times.

A *simple permutation* P_n is a simple arrangement $A_{n,n}$ of n distinct elements of class n, i.e., a tuple composed of n elements. Two permutations differ because of their order.

Given a finite multiset C of n elements with multiplicity $\alpha, \beta, \ldots$, a *permutation with repetitions* $P_n^{\alpha,\beta,\ldots}$ is a tuple composed of n elements, each of which can appear at most α, β, ..., etc. times. While simple permutations are a particular case of simple arrangements, permutations with repetitions are not a particular case of arrangements with repetitions. Permutations with repetitions differ, in addition to what holds for simple permutations, also in the case the elements are the same, but appear a different number of times.

A *simple combination* $C_{n,k}$ of n distinct elements of class k ("n choose k") is a subset composed by k out of the n elements ($0 \leq k \leq n$). Two combinations differ if there is at least a different element. The number of simple combinations is the number of simple arrangements divided by the number of permutations, i.e., the number of possible orderings, as ordering matters in arrangements, but not in combinations.

A *combination with repetitions* $C'_{n,k}$ of n distinct elements of class k is a subset composed by k out of n elements ($0 \leq k$) each of which can appear at most k times. Two combinations with repetitions differ, in addition to what holds for simple combinations, also in the case the elements are the same, but appear a different number of times.

Given a set C composed of n elements, its *power-set* $P(C)$ is the set of all subsets of C, empty set $\emptyset$ and set C included. If we interpret elements as choices, we are in the case where a solution is a taken or left labelling of the choices. Not all solutions are necessarily valid, an additional check keeps only the ones that satisfy the constraints, e.g., number of decisions, objective function, etc.

Partitioning a set C of n elements means identifying a collection $B = \{B_i\}$ of non empty subsets, often called blocks, such that each element belongs to one and just one block. The number of blocks k ranges from 1, i.e., there is one block that coincides with C, to n, i.e., each block is a unit set, i.e., it contains just one element of C. Ordering among blocks and inside each block doesn't matter, thus symmetries exist in partitioning. For example

$\{1,3\},\{2\},\{4\}$ and $\{2\},\{3,1\},\{4\}$ are some of the symmetrical partitions in 2 blocks of set $S=\{1,2,3,4\}$. For simplicity we do not consider partitions of multisets.

Powerset and partitioning often underpin problems that deal with combinatoric objects like sets, where the solution is one, some, or all of the set's subsets that satisfy some constraints. To the best of our knowledge, models supporting repetitions are not common in problem solving and the fact that k is no more upper bounded by n is not intuitive and students find it hard to understand how a solution can have a size bigger than the number of available choices.

4. Templates in C for Enumerative Combinatorics Schemas

A variety of languages and of paradigms is available to implement the Enumerative Combinatorics schemas described in Section 3. We propose for each of them a template in the C language based on recursion. As recursion is intrinsically a divide/decrease-and-conquer approach, the main issue is how to break into subproblems the generation of each element of the solution space. As discussed in Section 3, we envisage two avenues of attack:

1. The ith recursive call makes a decision on the i-th element of the solution, i.e., it picks one choice among the n available ones. The base case in general consists in verifying that we have reached the desired size k of the solution. Picking a choice requires an iteration that walks through the set of possible choices.
2. The ith recursive call decides whether to label each choice as either taken or left. The base case is reached when all choices are labelled. It may be necessary to check what has been found to make certain that it is indeed a solution.

We resort to the first one for all cases where the size k of the resulting set or multiset S is known, as it seems easier and more intuitive for students to understand: this is mainly related to a straightforward application of the principle of multiplication and a known height k of the recursion tree. For powerset and partition computations we use the second one, because k is not given and the taken or left labelling of choices is more natural.

4.1 Data Structures

Functions operate on either global information items or on local ones. Global means that the items are known, shared and accessible to all recursive calls, local means data created, generated or modified inside a recursive call, but not visible outside. Data structures like arrays, matrices, lists, tables, or other compound structures used for storing input data are normally global information items that recursive calls access: they typically operate in read/write mode on the proper array, list or table element, based on the recursion level or on other properly handled parameters. Global counters for recursive calls, solutions, etc. are other global data that should be visible and updated by all recursive calls. Depending on the adopted programming style or paradigm two options are available in order to handle global data structures:

- Using global variables of the C language, shared by all recursive calls. This is an easy and straightforward solution, but typically it is discouraged for the sake of program modularity.
- Handling global data as variables local to or dynamically allocated and initialized by the calling function, passed as parameters by reference across recursive calls.

We follow the latter option and we adopt an array-based representation of sets of information items. Information items are in general quite complex: in order to make our templates item-independent we map an information item onto an integer in a range, so that our templates internally work on integers and symbol tables implement the mapping. It is the same approach that [40] applies to graphs. Integers, not necessarily consecutive, thus serve the purpose of representing inputs and outputs of the problem, whereas the symbol table is in charge of input/output. A set or a tuple is represented as an array of integers, operated:

- As a stack, whenever it simply collects elements by adding or removing them through push/pop operations (type `Set` or type `Tuple` if order matters).
- As a direct access table, whenever Boolean or integer labels are associated to set elements, implemented as an array of integers.

A dynamic array, wrapped by a structure in the C language, is identified by the pointer `data`, by the number `n` of used elements and by the number `max` of allocated elements.

Declaration: Data structures

```
1. typedef struct { int *data; int n,
   max; } Set, Tuple;
2. typedef struct { Set *part; int n,
   max; } PartSet;
```

Both `Set` and `Tuple` types refer to the same base array type. We similarly introduce an array of sets `PartSet` for partitioned sets serving as inputs to the principle of multiplication template.

As our goal is not to store all solutions, rather to

Table 2. Schema-specific features

Schema	Base case	Iteration	Choice selection	Backtrack
multPrinc	lev >= k	i=0; i<curr->n; i++	push(S,curr->el[i]);	pop(S);

simplArr	S->n >= k	i=0; i<C->n; i++	if (mark[i] != 0) continue;
			mark[i]=1;
			push(S,C->data[i]);
				mark[i]=0;
				pop(S);

repArr	S->n >= k	i=0; i<C->n; i++	push(S,C->data[i]);	pop(S);

simplPerm	S->n >= C->n	i=0; i<C->n; i++	if (mark[i] != 0) continue;
			mark[i]=1;
			push(S,C->data[i]);
				mark[i]=0;
				pop(S);

repPerm	S->n >= nTot	i=0; i<C->n; i++	if (mark[i] <= 0) continue;
			mark[i]--;
			push(S,C->data[i]);
				mark[i]++;
				pop(S);

simplComb	S->n >= k	i=start; i<C->n; i++	push(S,C->data[i]);	pop(S);
repComb	S->n >= k	i=start; i<C->n; i++	push(S,C->data[i]);	pop(S);

list them one by one, we simply call function `checkAndPrint` that validates a given solution and possibly prints it. We also explicitly handle a counter `cnt` of generated and validated solutions, returned as a result by all recursive calls. Whenever necessary, we resort to the `lev` parameter to represent the recursive level of a given call. We omit constraints that define whether a choice is legal or not. They might be either global data or additional parameters.

4.2 Schemas in C

Algorithm 2 shows a multi-purpose template in C for all the Enumerative Combinatorics schemas of Section 3 for search problems. Optimization problems are not described for the sake of conciseness. They could be dealt with scoring the current solution and comparing it with current best one, instead of only checking the validity of the solution.

```
1. /* recursive function */
2. int functionR(...) { // function
   header
3.   int cnt = 0;
4. if (termination condition) {
5.     if (solution valid) {
6.       ...
7.       return 1;
8.     } else
9.       return 0;
10.  }
11.  for (iteration on choices) {
12.    make choice;
13.    cnt +=functionR(...) //
       Recursive call
14.    backtrack;
15.  }
16.  return cnt;
17. }
```

Algorithm 2. Multi-purpose template for Enumerative Combinatorics schemas

Each schema features its specific base case, iteration on choices, choice selection and backtrack, summarized in Table 2. For each schema we show the C statements for the function's header and for a recursive call. As already pointed out, optional parameters to handle best solutions in optimizaton problems are omitted.

As the *principle of addition* makes just one decision, the corresponding solution space is quite trivial. Its use is not common in our problem-solving context and thus we shall not consider it in the rest of this paper.

Principle of multiplication: the number of decisions is k and each choice is selected from a specific set. We limit the application of the principle of multiplication to problems modelled by disjoint sets. The data structure is thus a partitioned set C of `k=C->n` elements of type `Set`, as defined in Section 4.1. We reach the base case when `lev>=k`. Otherwise, if the `i`th choice is possible (`i<curr->n` where `curr=C->part[lev]`), we select it and we use it to extend the solution by 1 pushing `curr->data[i]` onto `S`. A linear recursive descent then starts on the next decision (`lev+1`). The integer variable `cnt`, used as the function's return value, stores the number of solutions. Backtracking is implemented as a pop from `S` operation.

```
int multPrinc(PartSet *C, Set *S, int
lev); //function header
multPrinc(C, S, lev+1); // example of
recursive call
```

The schema for simple arrangements is the root one, from which we derive the others. To the best of our knowledge, this is not very common in some contexts, e.g., the Anglo-Saxon world, where the basic schemas are permutations and combinations, whereas it is common practice in others, e.g., continental Europe.

Simple arrangements: the number of decisions is

k, the input is just one set of n choices and the output is a tuple of k decisions. The data structures are array C of type Set and array S of type Tuple. We apply here the principle of multiplication with overlapping choice sets, as choices at a given recursion level are the same as the ones at the previous level, with the exception of the last choice taken[3]. We reach the base case when S->n>=k. In the basic approach without pruning, we test the validity of the solution in the base case. In order to prevent using the same choice more than once an array of integers mark of size n records whether choice i has already been used (mark[i]=1) or not. This array is an example of how to impose a dynamic constraint on available choices. If we are not in the base case, if the ith choice is possible, i.e., i<C->n and not yet made, i.e., mark[i]==0, we select it, we mark it and we use it to extend the solution by 1 pushing onto S C->data[i]. A linear recursive descent then starts on the next decision. The integer variable cnt, the function's return value, stores the number of solutions. Backtracking is explicit: upon return from the recursive call, a pop occurs on S and mark[i] is reset to 0, making the ith choice available again.

```
int simplArr(Set *C, Tuple *S, int
*mark, int k); //function header
simplArr(C, S, n, mark, k);
// example of recursive call
```

Arrangements with repetitions: as repetitions of the same choice up to k times are allowed, there is no more need to check whether we have already picked a choice or not and thus there no need for array mark. Taking this remark into account, the code follows directly from the one for simple arrangements.

```
int repArr(Set *C, Tuple *S, int k); //
function header
repArr(C, S, k);
// example of recursive call
```

Simple permutations: they are simple arrangements where k=n. The code follows straightforwardly from the one for simple arrangements.

```
int simplPerm(Set *C, Tuple *S, int
*mark); //function header
simplPerm(C, S, mark);
// example of recursive call
```

Permutations with repetitions: this is the only case where the input is a multiset. Moreover we modify multiplicities during computation. For these reasons, we do not deem it necessary to introduce a specific type MultiSet. It is enough to have the base set C and an array for multiplicities mark, initialized with the number of instances of each distinct element in the base set. We pick choice i if mark[i]>0 and then we decrement mark[i]. This is what we mean by dynamically modifying multiplicities. Parameter nTot is the size of the multiset. The code follows the same strategy used for simple arrangements and simple permutations.

```
int repPerm(Set *C, Tuple *S, int
*mark, int nTot); //function header
repPerm(C, S, mark, nTot);
// example of recursive call
```

Simple combinations: they are derived form simple arrangements by "forcing" one of the possible orders. We walk through C with an index i initially set to start so we prevent reconsidering already used choices. The initial call to simpComb is with start assigned with 0. Recursion thus concerns both the following decision and the following choice (i+1). There are no constraints on the choices, thus array mark is not necessary. Taking this remark into account, the code follows directly from the one for simple arrangements.

```
int simplComb(Set *C, Set *S, int k,
int start); //function header
simplComb(C, S, n, k, i+1);
// example of recursive call
```

Combinations with repetitions: we proceed as in the case of simple combinations, the difference being that recursion concerns only the following decision and not the following choice (i unchanged). Index start is incremented only at the end of the for loop when no other choices are available.

```
int repComb(Set *C, Set *S, int k, int
start); //function header
repComb(C, S, n, k, i);
// example of recursive call
```

Powerset: we propose 2 implementations to compute the powerset of a set C of n elements based on:

(1) A strategy based on **element labelling**: every element can either be taken or left. We use arrangements with repetitions as a schema, provided we remember that the number of choices is 2 and that the number of decisions is n. The space size is $A'_{2,n} = 2^n$. The solution array S contains a 1/0 present/absent flag.
(2) A strategy based on **simple combinations**: the powerset is the union of the sets for simple combinations n choose i, where i is in the range

[3] Though not explicitly mentioned, we apply the principle of multiplication, following a similar pattern, in most of the subsequent recursive function templates.

from 0 to n. The C wrapper function is shown in Algorithm 3.

```
1. int powerset(Set *C, Set *S){
2.   int i, cnt = 0;
3.   for(k = 0; k <= C->n; i++)
4.     cnt += simplComb(C, S, k, 0);
5.   return cnt;
6. }
```

Algorithm 3. Implementation of powerset: wrapper function to iterate on simple combinations

The two templates are equivalent, but the first one is more oriented to enumerating the entire powerset, the second one is more useful in optimization processes where minimum- or maximum-cardinality subsets are requested, as the ascending or descending `for` loop may be stopped as soon as a suitable solution is found.

Set partitioning: in order to represent partitions we could:

- either indicate the unique block each element belongs to;
- or list the elements that belong to each block.

The first solution is preferable, as we use arrays of integers where each integer value represents the partition the element with that index belongs to. For example if $S = \{1, 2, 3, 4\}$, $n = 4$ and $k = 3$ (blocks have indices 0, 1 and 2), partition $\{1, 4\}$, $\{2\}$, $\{3\}$ is represented by an array `val` = {0, 1, 2, 0}.

Given a set C with n elements and a number of blocks k, partitioning C may require to find:

- any partition in exactly k blocks;
- all partitions in k blocks, where k ranges from 1 to n;
- all partitions in exactly k blocks.

If there is no need to identify symmetries or if just one solution is enough, we compute set partitions resorting to a generalization of the powerset implemented with the arrangement with repetitions mode. Instead of partitioning the elements in 2 subsets, i.e., the one belonging to the current subset and the ones not belonging to it, we put an element in one of the blocks whose labels range from 0 to $k - 1$. Since the definition of set partition excludes empty subsets, in the base case we test and prevent such a solution to be considered checking that each block contains at least one element. In the case of more sophisticated requirements we suggest to use Er's algorithm [37].

4.3 Searching for just one Solution

In many search problems there is no need to explore the whole solution space, as any of the solutions is enough. Preventing recursion from exploring the whole space is thus important, but the techniques for stopping it in an orderly way are not intuitive. Many students, especially those whose mental model of recursion is still rickety, believe they can "pierce" the sequence of recursive calls returning to the original calling function in one step. To avoid this, we present two templates:

- The first one is based on a flag, a global information item, implemented:
  - either as a global variable in the C language, though we strongly discourage this solution because of the reasons explained in Section 4.1;
  - or as a parameter passed by reference.
- The second one is based on a return value that is tested for success or failure.

Following the first approach, we declare an integer flag `stop` initially set to 0 (`int stop = 0;`) and we pass its pointer as a parameter to the recursive function (`functionR(... , &stop);`). In the recursive function:

- In the base case, if a valid solution has been found, the flag is set to 1 (`(*stop_ptr) = 1;`).
- In the iteration on choices, the loop iterating is controlled not only by the number of choices, but also by the flag still being at 0 (`for (i = 0; (`*iteration on choices*`) && (*stop_ptr)==0; i++)`).

Following the second approach, we have a recursive function with an integer return value (1 for success, 0 for failure):

- In the base case, if a valid solution has been found, the function returns success, else failure.
- In the the loop iterating on the choices:
  - we make a choice;
  - we test the result returned by the recursive function: if it is success, we return success;
  - when the loop is over, as no success case has ever been found, we return failure.

The multi-purpose template of Algorithm 2 is upgraded in order to accommodate our needs. Only the relevant parts are shown in the following code snippet.

```
1. /* main */
2. if (functionR(...) == 0)
3.   printf(''no solution
     found\n'');
4.
5. /* recursive function */
6. int functionR(...) {
7.   ...
8.   for (iteration on choices) {
```

```
9.      make choice;
10.   if (functionR(...))
11.     return 1;
        // Direct return upon success
12.   backtrack;
13.   }
14.   return 0;
15. }
```

Algorithm 4. Searching for just one solution: testing a success or failure return value

4.4 Improving Scalability

If we remain in the domain of complete algorithms, no matter how fast computers are or will be, when the size of the solution space grows exponentially, only small- to medium-size problems can be solved resorting to Enumerative Combinatorics in its basic version. However, for search problems, not all elements in the solution space are valid solutions, but only the ones that satisfy some constraints. In the basic templates seen in Section 4 we checked constraints in the base case, working directly on the solution, with or without auxiliary data structures. Of course this is a major waste of time and of computing resources, as we first compute and then discard many solutions. An alternative is to subordinate recursive calls to constraint satisfaction. We prevent exploration of a portion of the solution space where no valid solution can be found. Auxiliary data structures may or may not be needed. "*Pruning is the technique of cutting off the search the instant we have established that a partial solution cannot be extended into a full solution*" [33].

A constraint or validity condition consists in most cases of a logical condition, possibly tested in an intermediate node of the recursion tree, rather than in a leaf. There is no general-purpose methodology for pruning, however there are some frequently found cases:

- We enforce a *static filter* on available choices: in the iteration on choices, an acceptance constraint is logically conjuncted in the loop iteration condition. Acceptance constraints do not depend on previous choices, but only on the problem. Examples are boundaries on maps, available cells on paths, etc.
- We enforce a *dynamic filter* on available choices: in this case the acceptance constraint depends on past choices, i.e., on the current state. Examples are the location of chess pieces on the chequerboard that prevent some moves, the path already walked through in a maze, etc.
- We validate a *partial solution*, evaluating if there is any *hope* of eventually finding a valid solution or not. If a sufficient condition to decide that a solution cannot eventually be reached holds, we exclude whole regions of the solution space from further exploration.

Evaluating dynamic filters or validating partial solutions often require dynamic data structures, updated in the backtracking phase. As there is no general methodology, we introduce pruning by means of numerous examples. We require students to at least prune the solution space by subordinating recursive descent to constraint satisfaction. More advanced pruning techniques are highly appreciated both in assignments and in written exams.

Branch and Bound [41] for optimization problems, heuristic search methods, and approximate algorithms are beyond the scope of the course. Dynamic programming is on the contrary included in the syllabus, but it is beyond the scope of this paper.

4.5 Examples of Problems

Examples in the classroom, lab assignments and exam essays cover a wide range of problems, starting from simple ones up to NP-complete and NP-hard ones. An exhaustive list is out of scope, however some remarks help to elucidate our work.

Looking at their domains, the problems we deal with may be classified as:

- *Games and puzzles*: the 8-queens, Sudoku, the knight's tour, verbal arithmetic, tiling puzzles, etc.
- *Set and string theory*: set cover, longest increasing sequence, longest alternating sequence, etc.
- *Combinatorial optimization*: knapsack, weighted activity selection, minimal or maximal subsets satisfying a constraint, etc.
- *Graph theory*: clique, independent set, vertex cover, longest paths, feedback vertex or edge set, etc.

Looking at the underlying combinatorial model:

- Some problems exhibit a unique model, e.g., anagrams and permutations.
- For other problems there exists a panoply of applicable models, with varying outcomes in terms of efficiency.

The n-queens problem, for instance, belongs to the second group: queens can be undistinguishable, thus their order doesn't matter, or distinguishable. We could, in decreasing size of the solution space:

- Place or not place an undistinguishable queen on any of the n^2 cells (powerset implemented as arrangements with repetitions).
- Choose the n cells where to place a distinguish-

able queen out of the n^2 possible cells (simple arrangements).
- Choose the n cells where to place an undistinguishable queen out of the n^2 possible cells (simple combinations).
- Shrink the 2-dimensional board to one dimension (an array of n column indices), as each row must contain exactly 1 distinguishable queen (arrangements with repetitions, i.e., choosing tuple of n column indices).
- As each row and each column must contain exactly one distinguishable queen, permute the n different column indices (simple permutations).

Looking for a Hamiltonian path on an undirected and connected graph may be reduced:

- To computing simple permutations of the nodes, verifying for each of them in the base case that an edge from a node to the following one exists for all pairs of nodes.
- To applying a dynamic version of the principle of multiplication. The choices are the edges insisting on each node. It is dynamic because the cardinality of the choice set is not fixed at a given recursion depth. The approach is also introduced as a variant of permutations with pruning, given by the available edges.

For these classical examples students analyse complexity, in terms of solution space size, realizing the impact the choice of the model has on scalability, For the n-queens problem solution space size ranges from 2^{n^2} (powerset implemented as arrangements with repetitions, feasible only for very small values of n) down to $n!$ (simple permutations). For the Hamilton path, using the principle of multiplication walking only on allowed edges introduces a very efficient, though not easily estimated, pruning.

Taking models into account, combinations and permutations are commonly used whereas arrangements, both simple and with repetitions, appear less frequently. Whenever pruning is applied, a certain flexibility in the choice of the reference model or template is available, given the similarities between combinations and powerset, and between arrangements and permutations.

For strings and numbers order matters and arrangements are a convenient model when the goal is to identify subsequences of known length[4]. If the length of the desired subsequence is unknown, but an objective function must be satisfied, it is necessary to compute a sorted powerset and we suggest to resort to arrangements with repetitions. Some of the problems of this kind are well-known in the literature, like the *longest increasing sequence*, the *longest alternating sequence*, etc.

A wide range of problems' results can be represented as subsets. Whenever the cardinality of the subset is known a priori, as in sets order does not matter, simple combinations or combinations with repetitions fit as models. Whenever cardinality is unknown, all the subsets must be considered and the model is the powerset. If the objective function aims at finding a subset with minimum or maximum cardinality, the implementation of the powerset with combinations supports early stopping of the search, thus pruning. Many problems from graph theory fall in this category when the goal is to identify a minimal or maximal subset of the nodes or of the edges satisfying a certain validity condition. Examples are maximal clique, maximum independent set, minimum vertex cover, etc.

We find it useful to introduce combinatoric models as a starting point for state-of-the-art algorithms. Some examples are:

- Simple paths in directed or undirected graphs, and minimum distance problems: we interpret these problems as an application of permutations or arrangements with pruning, as they can be reduced to enumerating ordered sequences (tuples) of nodes.
- Minimum-spanning trees in weighted, undirected and connected graphs: they are a subset of the edges where both a validity constraint, i.e., spanning all nodes without loops and an optimality one, i.e., minimizing the sum of the weights of the selected edges, hold. The subset can be trivially generated by following the powerset template, or more like efficiently by the principle of multiplication on a partition of edges. Edges are partitioned based on the node they insist on. In our experience students' understanding of the problem increases, so they easily evolve to optimal greedy algorithms like Kruskal's and Prim's algorithms.
- We also solve clique, connected and strongly connected component problems by resorting to powerset models, properly enhanced with pruning.
- For both the longest increasing sequence and for the discrete knapsack problem: we are looking for a subset of the objects, the combinatorics model is thus the powerset. Starting from this basic understanding, students arrive more easily to the solution based on dynamic programming.

5. Educational Results

In this section we analyze some relevant statistical

[4] A subsequence may be obtained by deleting 0 or more non necessarily contiguous elements from the original sequence without affecting the order of the remaining elements.

data that we collected along the years in order to assess the impact of this approach to problem solving on the students. We consider the expectations faculty has about students' knowledge and skills, the perception students have of their knowledge and of their ability to apply it and results at exams. The course in its current format first appeared in the 2011/12 syllabus. In academic year 2013/14 we introduced schema-based instruction for problem solving based on Enumerative Combinatorics. In the next year we offered for the first time the written exam with two programming tracks (see Section 2.1). In academic year 2015/16 we published the textbook [14]. In academic year 2017/18 we retuned all lab assignments to make them fully compliant with the approach.

5.1 Faculty Expectations in Terms of Students' Knowledge and Skills

We did not expect that, after introducing our approach, most students would suddenly fully understand the new contents, apply the approach and improve all their skills. We rather expected a continuous improvement due to the joint effect of our actions in terms of contents delivery, availability of material and student support described in the previous paragraph.

Table 3 reports how we expected the students' receptiveness to course topics would evolve along the years. Scores close to 5 are for knowledge and skills any student should master at the end of the course. Intermediate scores around 3 characterize more advanced topics taught in the course, more challenging to master for the larger part of the audience. Lower scores are for topics we deemed only few motivated and capable students could master. Scores are average marks given by the faculty.

As regards to knowledge, the first 6 rows refer to the models. As regards to skills, in the following 6 rows we consider the ability to identify the base case, to iterate on choices, to stop as soon as a solution is found, to handle function calls, return values and parameter passing, to manage the backtracking phase and to prune the solution space.

In the earlier years we were satisfied when students used simple models, like the principle of multiplication or the powerset, identifying the correct base case and properly iterating on choices. In more recent years we expected proficiency in more advanced aspects like:

- The use of all the models, including partitioning.
- The introduction of pruning: efficient pruning is requested in simple cases and contributes significantly to the final mark in more difficult ones.
- The use of branch and bound techniques or the abilty to break symmetry: in the course we do not emphasize them, but we expect brilliant students to discover and use them on their own.

Our expectations define each year the threshold for passing the exam and the criteria for ranking. As data in Section 5.3 show, there has been a steady increase in the number of students that pass the exam. What Table 3 shows is that the level in terms of knowledge and skills required to pass the exam has also increased, thus combining quantitative and qualitative improvements. We believe we have now reached a steady state: there is no room in the syllabus for new topics given the number of credits allotted to the course and we can't further increase the threshold for passing the exam. Expected scores for academic years 2018/19 and 2019/20 are the same.

5.2 Students' Perception Of The Schema-Based Approach to Problem Solving

Starting from academic year 2016/17, the lab assignments described in Section 2.1 are discussed and ranked during a personal interview. One of the

Table 3. Expected students' receptiveness to course topics year by year

		2012/13	2013/14	2014/15	2015/16	2016/17	2017/18	2018/19 2019/20
Models	Mult. Princ.	4.3	4.3	4.3	4.4	4.4	4.4	4.4
	Powerset	4.0	4.0	4.1	4.3	4.3	4.4	4.4
	Arrangements	2.1	3.0	4.1	4.1	4.3	4.3	4.3
	Permutations	2.1	3.0	4.1	4.1	4.3	4.3	4.3
	Combinations	1.3	2.3	3.3	3.5	3.9	4.0	4.0
	Partitioning	0.1	0.8	2.4	2.4	2.7	2.7	2.9
Skills	Base case	3.7	4.2	4.2	4.5	4.5	4.5	4.5
	Choices	3.7	2.9	3.0	3.0	3.5	3.7	4.0
	Early stop	2.8	3.7	3.5	4.0	4.2	4.4	4.4
	Calls	1.2	1.5	1.5	2.4	3.2	3.6	4.2
	Backtrack	2.8	4.2	4.2	4.3	4.4	4.4	4.4
	Pruning	1.2	1.3	2.8	3.5	3.5	3.7	3.7

questions asked concerns the usefulness of schema-based instruction in problem solving. The very large majority of interviewees (>90%) agrees that the approach helps them in understanding the problem and that, applying the methodology, solving the problem is much easier. This has confirmed our impression that schema-based instruction was useful, though we could not support the claim with data or a detailed analysis.

To measure how students perceive schema-based instruction, at the end of the current 2019/20 winter semester we asked them to answer a questionnaire. During the lectures, we explained the students that we were collecting data for this paper and asked for their cooperation. Out of 424 new students, i.e., students attending the course for the first time, 2/3 answered, i.e., almost all those who were attending the lectures and handing in the assignments in due time.

The questionnaire breaks down in 4 sections asking students to self-assess:

1. Previous knowledge on Combinatorics and recursion from High School or from personal study, and its usefulness for the course.
2. Identification knowledge (Enumerative Combinatorics model selection), elaboration knowledge (adaptation of the model to the problem), planning and execution knowledge (algorithm design and coding in C).
3. Personal understanding of models and ability to apply them.
4. Skills related to recursion-based programming.

Sections 3 and 4 match the rows in Table 3, but this time from the student's perspective. Score 0 means no knowledge, useless or unable to apply, score 5 deep knowledge, extremely useful or perfectly able to apply.

As regards to Combinatorics, 63% of the students have previous knowledge. Note that Combinatorics is an optional topic in Italian High School Maths syllabuses. As regards to recusion, 75% of the students have no previous knowledge, 10% have some knowledge thanks to personal study and only in 15% of the cases recursion appeared in their High School syllabus, mainly in technical tracks (''Istituti Tecnici'').

Fig. 1 shows self-assessment results for previous knowledge of Combinatorics and recursion and its usefulness for the course. Only students that declared they have such previous knowledge are considered.

The mean is around 3 for the quality of previous knowledge both for Combinatorics and for recursion. The usefulness of recursion is scored again around 3. The usefulness of Combinatorics is lower (2.67), probably because at High School the approach was more Maths-oriented, in particular the goal was to serve as a basis for Probability Theory and not solution space modelling in problem solving. A standard deviation around .75 indicates that the values are reasonably close to the mean.

Taking into account all students, including those that have never been previously exposed to Combinatorics and recursion, Fig. 2 shows a drastic decrease in the scores and a very large spread in the values, especially for recursion, due to the large number of zeroes (no previous exposure). This proves that both Combinatorics and recursion are novel topics first addressed in a complete and systematic way in this course, no matter how technical the High School track might have been.

As regards to the degree students believe they have in identification, elaboration and planning/execution knowledge, Fig. 3 shows that the difficulty is perceived in identifying the model, in adapting and in implementing it. It also summarizes the very good opinion the students have about the usefulness of the schema-based instruction approach. This evidence confirms the qualitative impression collected during personal interviews.

The line chart of Fig. 4 shows how students rank their comprehension of the models and their ability to put them in practice. It also shows our expectations for the current academic year (see Table 3 rows labeled Models, last column).

Students confirm our classification of the models

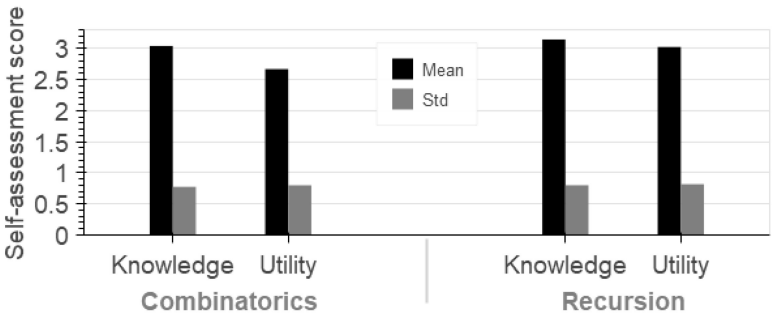

Fig. 1. Evaluation of previous knowledge of Combinatorics and recursion and of its usefulness (only exposed students).

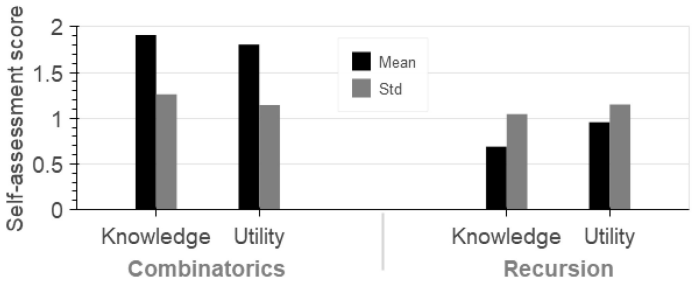

Fig. 2. Evaluation of previous knowledge of Combinatorics and recursion and of its usefulness (all students).

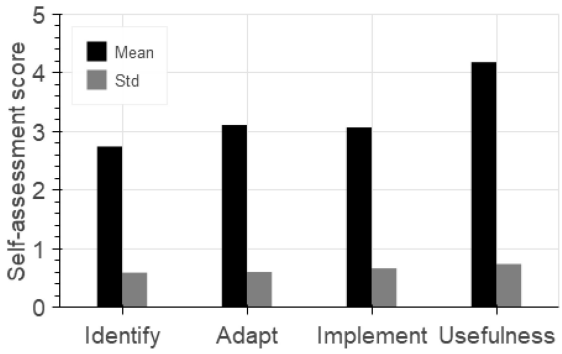

Fig. 3. Evaluation of identification, elaboration and planning/execution knowledge and usefulness of the approach.

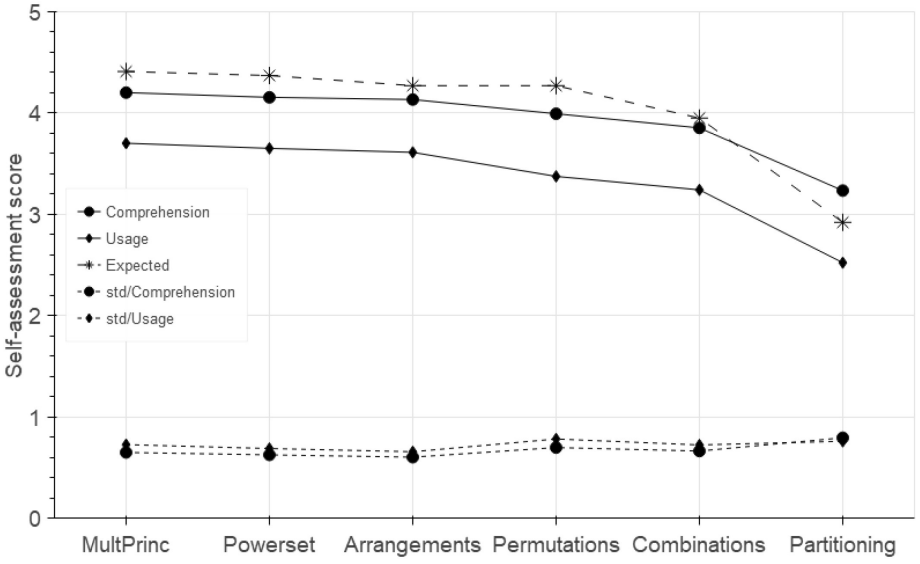

Fig. 4. Expected and perceived comprehension of models and ability to use them.

in terms of what is more or less difficult to understand. There are two exceptions:

1. In our opinion, permutations are as difficult as arrangements, students find them more difficult.
2. In our opinion, partitioning is much more difficult than what students perceive.

As regards to permutations, this is probably due to the reduced presence of permutations in examples and exercises with respect to arrangements. As regards to partitioning, as we believe it is difficult, examples and exercises we propose are simple, so students do not feel its complexity in general.

No surprise that understanding is easier than applying: the gap remains roughly the same for the multiplication principle, the powerset and arrangements, it widens for permutations and combinations, its maximum is for partitioning, proving that the students' perception of partitioning was fallacious. Standard deviation is largely under 1, witnessessing an acceptable uniformity in the overall evaluation by students.

Fig. 5 shows how students perceive their under-

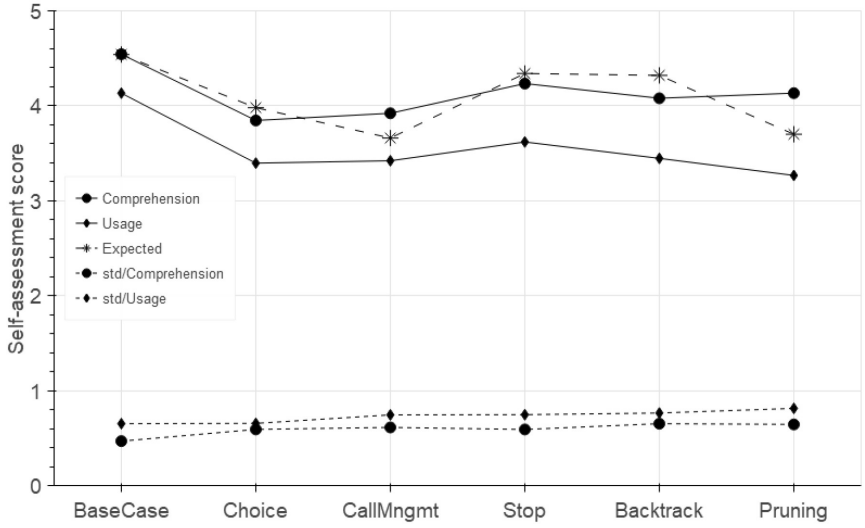

Fig. 5. Expected and perceived comprehension of key aspects of recursion and ability to use them.

standing and their skills in key aspects of recursion. It also shows our expectations for the current academic year (see Table 3 rows labeled Skills, last column).

Excluding tha base case, that is at no surprise considered as the easiest aspect both to understand ad to manage, there is a certain uniformity on all other aspects, with a second high for stop criteria, somehow related to the base case. Bigger difficulties in both comprehension and usage seem to relate to iterating on choices, to managing function calls, to pruning and backtracking. The message to us is that the way we address these topics must be improved. The gap between comprehension and usage exhibits two areas: a smaller difference is seen for more standard and well encoded aspects, such as base case identification, iterating on choices and managing function calls, whereas the gap increases with the more skill-demanding ones, like early stop, pruning and backtracking, considered much easier to understand than to put in practice. In our opinion this is due to the lack of a general-purpose methodology/scheme, replaced by many examples. As regards to the expectations of faculty listed in Table 3, there is a good agreement with students' perceptions for base case identification, iteration on choices and early termination. The expectations of faculty are in the middle between students' expectations in terms of understanding and usage for call management and pruning. We interpret this as the faculty has a more realistic view than the students about comprehension and skills in call management and pruning. As regards to backtracking, the lower score on the students' side means that, no matter the time already devoted to it, there is still a considerable part in the audience that finds it hard to follow and internalize the many examples emulating how recursion proceeds. It's our task to find a better balance between fast-learning students that do not appreciate an overly pedantic explanation and the slower ones, for whom even the simplest steps must be explained.

5.3 Measurable Results at Exams

We did not only look at our expectations or at the students' perception, but we also tried to measure the impact of our educational choices in terms of the students' final results at the exams. Let N be number of new students and P the number of new students who successfully passed the exam within the academic year (before September 30). The success ratio is $\rho = \frac{P}{N}$. Setting as year 0 the academic year 2011/12, Table 4 shows the percentage increase or decrease δ of ρ with respect to the value measured in year 0, the actions taken that year and the available support.

Academic years 2011/12 and 2012/13 show no substantial difference, the slight decrease being statistically insignificant. In academic year 2013/14 we introduced the schema-based instruction approach to problem solving with Enumerative Combinatorics. The only material available were the transparencies used during the lectures. Despite the limited amount of material, this resulted in an increase of about 5% of the success rate with respect to year 0. In the following year we offered for the first time the written exam with 2 programming tracks (see Section 2.1). As a result the increase with respect to year 0 jumped to 18%. In academic year 2015/16 the textbook [14] became available and the increase climbed to 47%. Retuning the labs and offering a 3rd parallel class in 2017/18 had no major consequence. The success rate has remained stable

Table 4. Percentage success rate increase with respect to year 0

Academic year	δ	Action	Support
2011/12	0.00%	None	
2012/13	–1.31%	None	videolectures
2013/14	5.33%	introduction of models	transparencies
2014/15	18.05%	two programming tracks	
2015/16	46.97%		textbook
2016/17	53.55%		
2017/18	45.76%	lab retuning/3rd parallel class	
2018/19	46.36%		

around 50% in the following academic years with some oscillations.

5.4 Graduate Students' Opinion

No data are available as regards to the opinion of graduate students on the course as a whole and of the schema-based instruction approach. However we receive many messages from graduate students we are still in touch with that state that the topics dealt by the course have been very useful in the sequel of their education. In particular the ability to apply recursion to problem solving seems to be a key point in job interviews. Graduates exposed to schema-based instruction have acquired a full mastery of the topics and in general perform quite well in interviews.

6. Discussion

Instead of leaving the students without guidance when solving a problem and writing code in the C language for it, we perceived that schema-based instruction could be of help. We imported from Combinatorics notions student had learned for other purposes, mainly Probability Theory, and, resorting to Enumerative Combinatorics, we used them to model the solution space for the problems they were facing. We provided them with a methodology, namely a sequence of steps to follow, starting from specification analysis, to combinatorial model identification, and with a set of templates for the models written in the C language. What was left to them was, apart the initial decision phase, to identify the constraints for valid solutions and to implement them in terms of validity checking functions, to identify the objective function in the case of optimization problems and to implement it. Another aspect where student creativity plays a key role is scalability: efficient pruning can make the difference in terms of size of the problem the approach can deal with and also in terms of ranking the student's solution in an exam.

This approach has been gradually introduced over the years: we started with the models explained during lectures, then we added more and more self-learning material in terms of transparencies, textbook, solved and commented lab assignments and new assignments every year. Students' response has been in line with our expectations: more students reached with less difficulty the required level for passing the exam, the average knowledge level of students has increased over the years, so that we are now requesting more than in the past, e.g., evaluation criteria include aspects like pruning that we normally disregarded.

However, our task as lecturers has not finished yet. We see some limitations in the current approach, that represent possible future work:

- The teaching-learning pattern relies on lectures, labs and assignments. Other patterns that would require more lecturer manpower are impossible in our context of bounded resources. However, we could develop self-assessment software in terms of question & answers, self-correcting exercises where the student has to complete partially written code. Existing experience however shows that it is difficult to offer other than simple examples.
- Students are more motivated when they feel that what they are learning has practical aspects. Alongisde with labs and assignments, we could envisage to start an in-course problem-solving challenge for some selected groups of volunteer top students. Limiting the number of groups also limits the manpower required to supervise them. The students may benefit not only in terms of knowledge and know-how, but also because this activity could complement or replace the final exam.
- Students have to make a leap of faith when we claim that problem solving and recursion are important for their their future career as Computer Engineers. We could gather examples of frequently asked questions in job interviews and relate them to recursion-based problem-solving. We could also be more precise in explaining the students the setting of our course in the context of the Computer Engineering curriculum, emphasizing the links with following courses. As it often

happens in academic environments, including Italy, each course is seen as monadic, i.e., stand-alone, independent and non communicating. Coordination with courses that precede and follow, though difficult to implement, would be of great value.

From the content's point of view, what we perceive as difficult or easy does not always match the students' feeling. In particular, we must revise the way we introduce permutations and pruning and focus more on partitioning, so that the students become aware of its intrinsic difficulty. We also need to focus more on the expression of constraints and of objective functions, not just in terms of a rich set of examples, but in more structured form of templates.

As regards to measurable results, analyzing questionnaires not resorting to professional support tools proved to be on the one hand heavy in terms of human resources devoted to this task and not accurate enough in statistical terms. However, though preliminary, it gave us a valuable insight in terms of matching faculty and students perceptions and expectations. Repeating an improved questionnaire in the editions to come with the support of better tools is a decision we have already taken.

7. Conclusions

Our long-standing experience in teaching a *CS2+CS3* course to second-year Computer Engineering students has allowed us to point out the main challenges students and lecturers encounter in learning and in teaching problem solving based on recursion.

There are difficulties inherent to recursion: it is an unnatural way of thinking, whose mode of operation is often understood in a wrong way, decentralized control, where each instance does its job, but there is no centralized supervision, is difficult to imagine, base case identification is not straightforward, apart from trivial cases, the recursive case is simple only if there is no selection among a panoply of choices, pruning strategies are needed to make it applicable to other than small-size problems.

There are difficulties inherent to problem solving: it is not mere exercise solving, it requires the ability to analyze a problem and the creativity to design a solution strategy. The need for creativity persuades many that all problems must be solved from scratch, as if it were the first time they appear. This is especially true in Informatics, a young science with respect to Mathematics, Logics, Physics etc. Consolidated methodologies are lacking and very often courses are like an artist's workshop: the lecturer is the artist, the students are the apprentices who look at what the artist does, try to imitate his/her work and then possibly to improve it.

No doubt creativity is a must in problem solving. However problems may be grouped into classes that share the same mathematical model. This is where schema-based instruction is valuable. It is top-down, rather than bottom-up: it requires to understand a concept and then derive consequences from it, rather than to gather lots of examples and then abstract. It promotes proactivity in students as regards to problem solving, it does not mean to turn them into repositories of passive knowledge.

We apply schema-based instruction to problem solving based on recursion. Schemas are models derived from Enumerative Combinatorics that describe in a formal and structured way a space where the solutions to a specific problem are located. We offer the students a complete portfolio of schemas in the form of templates written in the C language and a step-by-step methodology to approach problems and to design solutions. We do not avoid the need for creativity: creativity is still required in understanding the problem, in selecting the correct model, in identifying the conditions for a valid or for an optimal solution, in pruning the search space without loosing completeness. We rather harness problem solving within a clear framework.

We measure the effects of our schema-based instruction to recursion-based problem solving gathering data about the students' perception of the difficulties they encounter and compare them to our expectations. We also analyze improvements in successful exam completion and students' knowledge level.

These results show that schema-based instruction is a valuable approach in teaching and learning in general, beyond the original domains it was intended for, and that its version in terms of Enumerative Combinatorics models as C language templates has improved students' knowledge and know-how in recusion-based problem solving, their results at exams, the quality of our lectures and of our support material.

References

1. A. V. Robins, N. Rountree and J. Rountree, Learning and Teaching Programming: A Review and Discussion, *Computer Science Education* s.l.: Routledge, **13**(2), pp. 137–172, 2003.
2. R. Pessoa Medeiros, G. Lisboa Ramalho and T. Pontual Falcão, A Systematic Literature Review on Teaching and Learning Introductory Programming in Higher Education, *IEEE Transactions on Education* s.l.: IEEE, **62**(2), pp. 77–90, 2019.

3. E. Forcael, G. Garcés, P. Backhouse and E. Bastías. How Do We Teach? A Practical Guide for Engineering Educators, *International Journal of Engineering Education* **34**(5), pp. 1451–1466, 2018.
4. M. J. Prince and R. M. Felder, Inductive Teaching and Learning Methods: Definitions, Comparisons, and Research Bases, *Journal of Engineering Education*, **95**(2), 2006.
5. E. Lahtinen, K. Ala-Mutka and H. Järvinen, A study of the difficulties of novice programmers, *ACM SIGCSE Bulletin*, **37**(3), pp. 14–18, 2005.
6. L. E. Winslow. Programming pedagogy – A psychological overview, *SIGCSE Bulletin*, **28**(3), pp. 17–22, 1996.
7. S. P. Marshall. Schema-Based Instruction, *Encyclopedia of the Sciences of Learning*, Boston, MA.: Springer, pp. 2945–2946, 2012.
8. M. Chinnappan. Schemas and mental models in geometry problem-solving, *Educational Studies in Mathematics* s.l.: Kluwer Academic Publishers, **36**(3), pp. 201–217, 1998.
9. N. B. Dale, *Most Difficult Topics in CS1: Results of an Online Survey of Educators*, New York, NY, USA: ACM, **38**, pp. 49–53, 2006
10. M. Hertz and S. M. Ford, Investigating Factors of Student Learning in Introductory Courses, *Proceeding of the 44th ACM Technical Symposium on Computer Science Education,* New York, NY, USA: ACM, pp. 195–200, 2013.
11. E. S. Roberts, *Thinking Recursively: With Examples in Java*, USA: John Wiley & Sons, Inc., 2005.
12. E. Lee, V. Shan, B. Beth and C. Lin, A Structured Approach to Teaching Recursion Using Cargo-Bot, *Proceedings of the Tenth Annual Conference on International Computing Education Research*, New York, NY, USA: ACM, pp. 59–66, 2014.
13. W. Dann, S. Cooper and R. Pausch, *Using Visualization to Teach Novices Recursion*, New York, NY, USA: ACM, **33**, pp. 109–112, 2001.
14. G. Cabodi, P. E. Camurati, P. Pasini, D. Patti and D. Vendraminetto, *Ricorsione e problem-solving. Strategie algoritmiche in linguaggio C.* s.l.: Apogeo Education, 2015.
15. A. Vihavainen, J. Airaksinen and C. Watson, Systematic Review of Approaches for Teaching Introductory Programming and Their Influence on Success, *Proceedings of the Tenth Annual Conference on International Computing Education Research.*Glasgow, UK: ACM, pp. 19–26, 2014.
16. M. W. van Someren, Y. F. Barnard and J. A. C. Sandberg, *The Think Aloud Method: A Practical Guide to Modelling Cognitive Processes.* s.l.: London: Academic Press, p. 208, 1994.
17. N. Arshad. Teaching Programming and Problem Solving to CS2 Students Using Think-Alouds, *Proceedings of the 40th ACM Technical Symposium on Computer Science Education* s.l.: ACM, **41**(1), pp. 372–376, March 2009.
18. C. Rinderknecht, A Survey on Teaching and Learning Recursive Programming, *Informatics in Education*, **13**, pp. 87–119, 2014.
19. J. Gal-Ezer and D. Harel. What (Else) Should CS Educators Know? *Commun. ACM* s.l.: ACM, **41**(9), pp. 77–84, 1998.
20. J. Gal-Ezer and E. Zur. What (Else) Should CS Educators Know?: Revisited, *Proceedings of the 8th Workshop in Primary and Secondary Computing Education*, New York, NY, USA: ACM, pp. 83–86, 2013.
21. D. R. Woods, A. Hrymak, R. R. Marshall, P. Wood, C. Crowe, T. W. Hoffman, J. D. Wright, P. A. Taylor, K. A. Woodhouse and K. C. G. Bouchard, Developing Problem Solving Skills: The McMaster Problem Solving Program, *Journal of Engineering Education* s.l.: American Society for Engineering Education, **86**(4), 1997.
22. R. McCauley, S. Grissom, S. Fitzgerald and L. Murphy, Teaching and learning recursive programming: a review of the research literature, *Computer Science Education* s.l.: Routledge, **25**, pp. 37–66, 2015.
23. S. B. McCalla, Greer J. E. and I. Gordon, Supporting the Learning of Recursive Problem Solving, *Interactive Learning Environments* s.l.: Routledge, **4**, pp. 115–139, 1994.
24. H. Kahney. What Do Novice Programmers Know About Recursion, *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, New York, NY, USA: ACM, pp. 235–239, 1983.
25. T. Götschi, I. Sanders and V. Galpin, Mental Models of Recursion, *SIGCSE '03 Proceedings of the 34th SIGCSE technical symposium on Computer Science Education*, New York, NY, USA: ACM, **35**, pp. 346–350, 2003.
26. J. H. Conway and R. Guy, *The Book of Numbers*, s.l.: Springer, 1996.
27. D. Dicheva and S. Close, Mental Models of Recursion, *Journal of Educational Computing Research*, **14**(1), 1996.
28. M. Petković, *Famous Puzzles of Great Mathematicians.* s.l.: American Mathematical Society, 2009.
29. S. Hamouda, S. H. Edwards, H. G. Elmongui, J. V. Ernst and C. A. Shaffer, RecurTutor: An Interactive Tutorial for Learning Recursion, *ACM Trans. Comput. Educ.*, New York, NY, USA: ACM, **19**, pp. 1:1–1:25, 11 2018.
30. B. S. Bloom, R. D. Krathwohl and B. B. Masia, *Taxonomy of educational objectives. The classification of educational goals. Handbook 1: Cognitive domain*, New York: Longmans Green, 1956.
31. G. Polya, *How to Solve It: A New Aspect of Mathematical Method.* Second. s.l.: Princeton University Press, 1957.
32. H. L. Rolf, *Finite Mathematics*, s.l.: Brooks/Cole, 2013.
33. S. S. Skiena, *The Algorithm Design Manual*, 2nd. s.l.: Springer Publishing Company, Incorporated, 2008.
34. J. S. Kimmel, H. S. Kimmel and F. P. Deek, The common skills of problem solving: From program development to engineering design, *International Journal of Engineering Education*, **19**(6), pp. 810–817, 2003.
35. M. G. Voskoglou and A. B. M. Salem, Analogy-Based and Case-Based Reasoning: Two sides of the same coin, *International Journal of Applications of Fuzzy Sets and Artificial Intelligence*, **4**, pp. 5–51, 2014.
36. R. P. Stanley, *Enumerative Combinatorics*, 2. s.l.: Cambridge University Press, **1**, 2011.
37. M. C. Er, A Fast Algorithm for Generating Set Partitions, *The Computer Journal*, **31**, pp. 283–284, 1988.
38. A. Repenning, A. Basawapatna and N. Escherle, Computational thinking tools, *Proceedings of the IEEE Symposium on Visual Languages & Human-Centric Computation*, Cambridge, UK: s.n., pp. 218–222, 2016.
39. R. P. Grimaldi, *Discrete and Combinatorial Mathematics; An Applied Introduction*, Boston: Addison-Wesley Longman Publishing Co., Inc., 1985.
40. R. Sedgewick, *Algorithms in C, Part 5: Graph Algorithms, Third Edition,* s.l.: Addison-Wesley Professional, 2001.
41. A. G. Doig and A. H. Land, An automatic method for solving discrete programming problems, *Econometrica*, s.l.: Wiley, Econometric Society, **28**, pp. 497–520, 1960.
42. G. E. Martin, *Counting: The Art of Enumerative Combinatorics*, s.l.: Springer New York, 2001.
43. M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*, New York, NY, USA: W. H. Freeman & Co., 1990.

Gianpiero Cabodi is associate professor at Dipartimento di Automatica e Informatica, Politecnico di Torino (Italy). He graduated in 1984 in Electronic Engineering and received his PhD in Computer and System Engineering in 1988. His main fields of interest include formal methods, model checking, binary decision diagrams, Boolean satisfiability, logic and high level synthesis, verification problems in cybersecurity.

Paolo Enrico Camurati is full professor at Dipartimento di Automatica e Informatica, Politecnico di Torino (Italy). He graduated in 1984 in Electronic Engineering and received his PhD in Computer and System Engineering in 1988. His current research interests include formal verification of hardware correctness and verification problems in cybersecurity.

Paolo Pasiniis a Post-Doctoral Researcherat Dipartimento di Automatica e Informatica, Politecnico di Torino (Italy). He graduated in 2012 in Computer Engineering and received his PhD in Computer and Control Engineering in 2017. His main fields of research include formal methods, formal verification of hardware correctness and model checking.

Denis Patti received both the MS degree in computer engineering and the PhD degree in computer and control engineering from Politecnico di Torino in 2012 and 2018, respectively. He is currently a post-doc researcher at Dipartimento di Automatica e Informatica of the same University.

Danilo Vendraminetto is a Post-Doctoral Researcher at Politecnico di Torino. He has a PhD degree in computer engineering from the same Institution. His research interests include formal methods applied to verification of design correctness, model checking, and cybersecurity in embedded systems.