Programming Education Across Disciplines: A Nationwide Study of Danish Higher Education

Sebastian Mateos Nicolajsen[1*], Sofie Nielsen[1], Louise Meier Carlsen[1], Claus Brabrand[1]

[1*]Center for Computing Education Research, IT University of Copenhagen, Rued Langgaards Vej 7, Copenhagen, 2300, Denmark.

*Corresponding author(s). E-mail(s): sebni@itu.dk;
Contributing authors: soni@itu.dk; loca@itu.dk; brabrand@itu.dk;

Abstract

Decades of technological development and innovation have led to an unprecedented digitalization of society. Graduates entering the modern workforce now need better computational competences. Higher education is thus forced to adapt and consider how to support these demands. To support educators in making decisions regarding *how* to integrate computing in their programmes, we set out to investigate to what extent and how mandatory (computer) *programming* has emerged in the tertiary educational system of an entire country: Denmark. We analyzed all course descriptions from 1,169 tertiary educational programmes at all higher educational institutions spanning all of Denmark. This provides a "snapshot" of where and how programming education has emerged and manifested itself in higher education across faculties, study programmes, and disciplines for an entire country.

Our results demonstrate that, as of 2023, 1 in 6½ educational programmes (175 out of 1,169) include mandatory programming. To support educators in introducing programming, we quantify and provide an overview of educational programmes with mandatory programming along several dimensions. We characterize the roles programming has in different programmes, how programming is delineated, and which families of competences are often taught in connection with programming. Based on the results of our study, we issue five recommendations directed at policymakers and educators responsible for navigating the inclusion of programming in their education.

Keywords: Educational programmes, Compentencies, Programming, Computing.

1 Introduction

Most occupations today involve computers and require knowledge of the digital realm. Computing education is, to many, a subject that ought to be part of core curricula to ensure sufficient digital competences for the next generation (Gal-Ezer and Stephenson, 2014; Menekse, 2015). Already in 1985, Haigh (1985) argued for the importance of education in computing. As put by Jones[1] et al. (2013):

> *Our young people should be educated not only in the application and use of digital technology, but also in how it works, and its foundational principles. Lacking such knowledge renders them powerless in the face of complex and opaque technology, disenfranchises them from making informed decisions about the digital society, and deprives our nations of a well-qualified stream of students enthusiastic and able to envision and design new digital systems* (Jones et al, 2013).

However, to do so, we must understand the different needs of different disciplines—a student wanting to pursue a career in biology will likely need computing for different reasons and to a different degree than a student pursuing a career in archaeology (Denning and Tedre, 2021; Evrard and Guzdial, 2023). Currently, we have no broad understanding of these differences on a curricula level, let alone at a national scale. Consequently, educators are left in the dark with little knowledge of *why* computing should be included, *what* should be included, and *how* one delineates and implements it into a higher education programme for a given discipline (Caspersen et al, 2017; Palumbo, 1990).

The objective of this paper is to examine how computing has organically emerged across all disciplines in higher education. We address these issues from a pragmatic, application-oriented perspective. Paraphrasing Evrard and Guzdial (2023), the answers to these lingering questions lie in the individual disciplines.

We have identified 175 Danish tertiary educational programmes that have at least one mandatory course introducing (computer) programming. (This comprises 130 distinct courses; less than 175 programmes because some courses are shared across multiple educational programmes.) We analyse the course description curricula of these courses to identify and taxonomize how different disciplines go about teaching and introducing mandatory programming.

Our results indicate that 1 in 6½ programmes include mandatory programming. The sheer magnitude of this number (15% as of 2023) renders programming relevant for higher education, in general. The remaining educational programmes are likely to, sooner rather than later, find themselves in a situation where they need to equip their students with programming competencies beyond simply *using* a computer. For these reasons, it is our intention to help enable policymakers and educators to take ownership of *how* programming should be included (or *why* it should be excluded) in their educational programmes and make sure they are able to make such decisions in an informed manner. We address the following research questions to support and inform educators on these topics:

RQ1 (Programmes): *What kind of educational programmes introduce programming?*

[1] Simon Peyton Jones, chair of Computing At School (Association for Computing education in the UK).

RQ2 (Role): *What role does programming play in non-computing disciplines?*

RQ3 (Delineation): *How is programming delineated in non-computing disciplines?*

RQ4 (Competences): *What kind of programming competences appear across disciplines?*

We focus on *Danish* tertiary education for two reasons. First, computing education is still in its infancy and not yet mandatory in Danish primary nor secondary education, leaving it up to tertiary educational institutions to create their own curricula (Bundsgaard et al, 2019; Caspersen et al, 2017). In Denmark, tertiary educational curricula are created in collaboration with industry partners through 'employer panels' to help guarantee industry-relevant competences (De Raadt et al, 2002). As such, educational programmes in Denmark can be expected to implement computing competences *relevant* to their disciplines. Second, Denmark is a relatively small country with a population of 5.9M and has a sufficiently limited number of tertiary educational institutions (universities, university colleges, and business academies) which makes it feasible to conduct a comprehensive national study. Thus, studying the case of Denmark allows us to explore *all* disciplines for an *entire* country and create a comprehensive national "snapshot" exhibiting the state of the emergence of programming in higher education. Interestingly, this "snapshot" was taken right before the widespread adoption of generative artificial intelligence (GAI), which will likely disrupt the educational landscape in the future.

2 Background

During the last decade, computing education has been gaining global momentum; in particular, in response to Wing's (re-)introduction of *'Computational Thinking'* (Wing, 2006). This has spawned many movements; e.g., *'Computing for All'* and *'Computing at School'* (Caspersen et al, 2017; Crick and Sentance, 2011). Today, computing can be argued to be a crucial part of twenty-first-century skills (Gretter and Yadav, 2016). Further, in some countries, e.g., Ireland, higher education is expected to enhance its contributions regarding the computing competences of graduates (Stephens et al, 2007).

The computing education research discipline has thus provided a wealth of material to support relevant education and teaching (Valentine, 2004). The following sections introduce the role of computing across disciplines, central competences, and the research on existing courses introducing *programming*.

2.1 Common Competences of Programming in Education

The inclusion of programming/computing in higher education has been significantly influenced by three concepts: *Computational Literacy* (DiSessa, 2000), *Computational Thinking* (Wing, 2006), and *Digital Competences* (Ilomäki et al, 2011). The concepts have been used to promote (political) focus on *computational empowerment*; i.e., enabling individuals to partake in a digital society (Nicolajsen et al, 2021). While distinct concepts, they promote similar skills and competences. For this reason, we briefly summarize three topics that *cross-cut* these concepts, namely: *Problem-Solving*, *Algorithmic Thinking*, and *Abstraction*.

Problem-Solving is concerned with *general* problem-solving and not how to solve *specific* "template problems"; e.g., being able to solve a variety of different problems (Lu et al, 2022; Robins et al, 2003). Some researchers argue that problem-solving should be the primary element to teach (Pears et al, 2007). Kalelioglu et al (2016) describe a five-step generic process for problem-solving (in computational thinking): (1) *identify the problem*, (2) *represent and analyse data*, (3) *select and plan solutions*, (4) *implement solutions*, and (5) *assess solutions*. Similar definitions of problem-solving exist specifically for (object-oriented) programming: (1) *problem identification and analysis*, (2) *design*, (3) *implementation*, and (4) *validation* (Madsen et al, 1993). Computational problem-solving is today, more often than not, used in collaboration with so-called *domain experts* and therefore enhances problem analysis, solution design, and implementation (Gal-Ezer and Stephenson, 2014).

Algorithmic Thinking (aka, *Algorithmic Problems & Solutions*) is, by many, perceived as the core of computing (Gal-Ezer and Stephenson, 2014; Lu et al, 2022). Algorithms are often centralised based on the idea that algorithms are what computers execute. However, not only does algorithms allow the automation of actions, they can also act as a way of perceiving the world and theorize about its behaviour, with or without a computer (Curzon and McOwan, 2017).

Abstraction concerns itself with the idea of working with different levels of detail, being able to work with different layers of abstraction (Nicolajsen et al, 2021; Lu et al, 2022). Abstraction is also argued to be a core competence as it is central to developing computational solutions (Bennedsen and Caspersen, 2006).

2.2 Current Role(s) of Computing

Until recently, computing has been perceived mostly as a *medium of expression* (aka, *Computing for Expression*); earlier, the computing education community has largely focused on development without consideration of much else (Ko et al, 2020).

Limitations of Computing. Researchers are now arguing that programming curricula should also consider *limitations* of computing and programming, including inherent biases. More specifically, Evrard and Guzdial (2023) propose two additional perspectives on computing. First, *Computing for Discovery* is the application of encapsulated automation, allowing *domain-expert* users to explore rich systems of composed models to expand theories; e.g., simulation (Evrard and Guzdial, 2023; Nicolajsen et al, 2021). Second, *Computing for Justice* (aka, *Critical Computing*) is the exploration of how computing affects culture, society, and politics. The argument by Ko et al. is supported by the findings of Barendsen and Steenvoorden (2016), who argue that societal aspects are not equally apparent in all curricula (based on analysis of curricula from the *Computer Science Teachers Association*, *Computing at School*, France, and The Netherlands). A different, and older, perspective on the role of computing comes from Denning et al (1989) wherein computing can either be viewed as 1) *discipline-oriented thinking*, i.e., the invention of new distinctions in the field, expanding vocabulary and enabling new modes of actions and the creation of tools for others to use, *or* for 2) *tool use*, i.e., being capable of using these modes of action for different purposes.

Type(s) of Learners. For education in computing, one cannot exclusively focus on *topics* and *competencies*. One should also consider the *learners*; in particular,

the diversity of novice learners in terms of their background and future endeavours (Robins et al, 2003). Specifically, Shaw (2022) returns to the idea of *vernacular developers*[2] taken as those who create software in a specific context wherein the software is a means to an end. Synonymously, such programmers are called *conversational programmers* (Evrard and Guzdial, 2023). Such individuals do typically not identify as "programmers," yet work with spreadsheets, scripts, and databases that frequently involve programming (Shaw, 2022). If we are to consider programming in all of higher computing education, the majority are likely to be such vernacular/conversational programmers. Thus, one cannot readily extrapolate conventional programming teaching to the more general context of diverse computing education, as the audience is seldom professional developers (where programming *is* an end in itself). This is further supported by research suggesting that tailoring introductory programming teaching is important to support interest in computing (Chakrabarty and Martin, 2018; Ott et al, 2018). Specifically, the education should be augmented by engagement in scenarios which meet students' interests, talents, and career goals (Thiry et al, 2011; Christensen et al, 2021).

2.3 Related Work and Introductory Programming

While current ideas of computing education discuss the widespread applicability of computing, one should look towards the courses of educational programmes that apply it in practice.

Several publications report on implementing introductory programming courses in higher education both inside and outside of computer science. Luxton-Reilly et al (2018) has created an extensive review of these. They highlight a curriculum focus on topics such as *UML modelling*, *design patterns*, *problem-solving*, and *algorithms*. Further, they identify courses bridging the gap between introductory programming and other disciplines, such as biology or law. Finally, they found that such publications provide little detail about the context of the course and advocate that future publications embed such contextual information to facilitate the application of findings.

Becker and Fitzpatrick (2019) explore what educators expect of higher education introductory programming students globally. They highlight terms applied in expected learning outcomes; e.g., *programming*, *design*, *data*, *algorithms*, *testing*, *methods*, and *functions*.

Further, there appears to be a large focus in literature to identify the programming language used in such courses in different countries and continents (Mason and Cooper, 2014; Aleksić and Ivanović, 2016; Murphy et al, 2016; Mason et al, 2024; Becker and Fitzpatrick, 2019).

In conclusion, there is plenty of work investigating how to develop courses introducing programming in higher education. Our present study, however, takes a different approach by exploring the curricula of courses introducing mandatory programming at all higher educational institutions for an entire country, namely Denmark.

[2] Often the term *end-user developer* is used as an alternative to *vernacular developers*, which has provided a wealth of laudable research (e.g., Ardito et al (2012)). However, as argued by Shaw (2022), this term may be pejorative.

8	Universities	University of Copenhagen (KU), University of Aarhus (AU), University of Southern Denmark (SDU), University of Roskilde (RUC), University of Aalborg (AAU), Technical University of Denmark (DTU), Copenhagen Business School (CBS), IT University of Copenhagen (ITU)
8	Vocational Academies	Zealand Academy, Copenhagen Academy, Copenhagen Busniess Academy, IBA, SydVest Academy, Midtvest Academy, Aarhus Academy, Dania Academy
7	University Colleges	Copenhagen University College, Absalon University College, UCL Vocational Academy and University College, University College of Southern Denmark, VIA University College, University College of Nordjylland, The Danish Media and Journalist College
3	Art Schools	The Royal Academy, Architecture School of Aarhus, Design School of Kolding
26	**Institutions**	**at the tertiary educational level in Denmark**

Fig. 1: Overview of all tertiary educational institutions in all of Denmark.

3 Educational Context

In Denmark, there is (still as of 2024) no mandatory standalone computing subject in primary school. In secondary school, the one-year computing course *'informatik'* is only *mandatory* in two out of four types of high schools. In the most common type of high school (STX), attended by more than half of high-school students (Statistik, 2022), the course is only offered as an *elective* (Caspersen et al, 2017). For this reason, tertiary educational institutions have to implement basic programming teaching "from scratch" to students entering their educational programmes.[3] There are 26 tertiary educational institutions in Denmark (listed in Figure 1). These institutions offer a variety of different educational programmes. We focus on the *professional bachelor*, *bachelor*, and *master* degree programmes which comprise 3½, 3, and 2 years of full-time studies, respectively (corresponding to 210, 180, and 120 ECTS points[4]). We focus on these since institutions offering such degrees have sole ownership of the so-called *competence profiles* of these educational programmes, which are designed in collaboration with institution-chosen *employer panels*.

By law, institutions are required to publish an official curriculum, in the form of a so-called *course description*, for each course within an educational programme. Course descriptions follow one of two standards (exemplified in Figure 2). The course descriptions are required to specify *explicit learning goals*. The Danish grade scale defines grades as *the degree of fulfilment of explicit learning goals* (see Figure 14, in the appendix). Hence, if a course does not have learning goals, one cannot meaningfully issue grades. The first format specifies *explicit learning goals* in the form of Intended Learning Outcomes (ILOs, see Figure 2a); these are most often based on

[3]According to the official curriculum of the Ministry of Education, teaching in programming is limited to variables, sequences, loops, if-statements, and functions https://www.uvm.dk/-/media/filer/uvm/gym-laereplaner-2017/hhx/informatik-c-hhx-htx-stx-august-2017.pdf.

[4]ECTS is an abbreviation of European Credit Transfer and Accumulation System: 60 ECTS points corresponds to one full-time academic year of studies.

- ***Describe*** the concepts behind various mathematical approaches to finding numerical [...]
- ***Describe*** mathematical approaches in the form of pseudocode.
- ***Translate*** pseudocode into computer code.
- ***Use*** pseudocode to construct computer algorithms.
- ***Apply*** algorithms to examine the quality of numerical solutions to differential equations.

— [*Computer algorithms* - Biotechnology, AU]

(a) Intended Learning Outcomes (ILOs).

Knowledge:	Explain how a computer works and the process around programming computer-based information systems.
Skills:	Analyze and plan a programming process.
Competences:	Use a programming language. Make decisions about system configuration vs. adaptation.

— [*Software development* - IT, Communication & Organisation, AU]

(b) European Quality Framework (EQF).

Fig. 2: Examples of the two ways of phrasing explicit learning goals.

the SOLO Taxonomy (Biggs and Collis, 2014). Institutions adhering to this format generally subscribe to the principles of Constructive Alignment as their teaching philosophy (Biggs et al, 2022; Brabrand and Dahl, 2008, 2009). The second format follows the European Qualification Framework (EQF, see Figure 2b) and separates *explicit learning goals* into three categories: *knowledge*, *skills*, and *competences* (Méhaut and Winch, 2012). Some institutions (or individual courses) specify both ILOs as well as *knowledge-skills-competences.* The existence of explicit learning goals, for all courses, on all educational programmes, in an entire country, provides us with an opportunity for a study systematically harvesting and analyzing these. In addition to explicit learning goals, some of the course descriptions also include information about the content, structure, mandatory activities, exam format, literature, and the format of the teaching.

Most programmes offer flexibility in terms of electives, conversion courses, and student-chosen specialization courses. This is especially true for master programmes; bachelor and professional bachelor degrees are usually a lot more restrictive in terms of such individualization. Two institutions, however, stand out and are worth highlighting.

The Technical University of Denmark (DTU) allows students to design their own programmes, requiring students to have a minimum number of ECTS points in a given topic (each topic is typically restricted to a set of particular courses). To accommodate this, they offer the same courses (or very similar courses) during different times of the

week. Naturally, this leaves the students with many options to determine which courses and programming courses, they want to enroll in. In such cases, our study focuses on the *first* available mandatory introductory programming course listed. We disregard any educational programmes where students can *avoid* an introductory programming course; e.g., by enrolling in a different technical course.

The University of Roskilde (RUC) allows students to combine multiple types of degrees; e.g., computer science *and* psychology. Consequently, students may enroll in an introductory programming course given many different educational combinations. Choosing *computer science* (which is one of multiple technical tracks), allows for 13 different educational programmes.[5] We count this as a single educational programme, but the reader should be aware of the flexibility available at this institution.

Lastly, any educational programme with a single elective allows for students to enroll in an introductory programming or computing course; e.g., a student enrolled in a history degree at the University of Copenhagen (KU), can enroll in an introductory programming course at the computer science or mathematics department.[6] We recognize that such choices will greatly affect the individual competence profile of the student; however, it is not mandatory and does not reflect an institutional (or employer) perceived need for such competences. More strongly, the inclusion (and requirement) of a specialization, e.g., a set of connected "electives," implies a higher degree of institutional recognition of certain needs within the industry. Yet, as these are still "avoidable" by students, we have excluded specialization courses from this study. Some programmes rely on nationally (loosely) defined curricula for their programmes. We exclude such programmes since they offer no insight into the autonomous decisions of educational institutions. We thus adopt a conservative approach to considering courses and programmes. For this reason, our study highlights the competences deemed most important and essentially provides the *raison d'être* of programming in higher education; i.e., what course responsibles, head of study programmes, and (potentially) employer's panels find crucial for subsequent courses or for joining the workforce.

4 Methodology

The first author collected data on the majority of Danish tertiary educational programmes that introduce (computer) programming. Using the public list of Danish educational programmes,[7] he examined 679 tertiary educational programmes, some of which were offered by multiple institutions, yielding a total of 1,169 individual programmes. Programmes were only investigated if the list provided an overview of where the programme was offered, which excluded 209 programmes (not included in the number above). The data was gathered during the spring of 2023, and the courses gathered were either from the previous iteration of the course or, if available, the following iteration. Therefore, this study constitutes a "snapshot" of the state of mandatory programming emerging in Danish tertiary education. Educational programmes were included if, and only if, a course title, description, or curriculum

[5] Find the right Bachelor Education: https://ruc.dk/bachelor/kombinationer/129
[6] Bachelor in History: Education & Structure: https://studier.ku.dk/bachelor/historie/#undervisning
[7] The Education Guide: http://ug.dk

Step	%	Coding	Coder(s)
(1)	20%	*inductively establish* initial codes	A1
(2)	20%	*deductively validate* initial codes	A1
(3)	-	*refine* initial codes against literature	A1
(4)	20%	collaborative coding *for training*	A1+A2
(5)	20%	independent coding *for validation*	A2
(6)	80%	code remaining data (based on refined codes and collaborative coding)	A1

Fig. 3: Overview of the coding process. The steps above the line are for *producing* the codes; the steps below the line are for *investigating the reliability* of the codes produced.

contained words related to programming (`programming`, `coding`, *or* `scripting`) or mention software allowing programming or a specific programming language (e.g., `Python` *or* `R`). Such courses were identified by examining each programme's compulsory courses (and their description). Furthermore, only compulsory or conversion courses were included, as such courses underscore how an institution finds the competences associated with programming mandatorily indispensable. We have made our data set publicly available.[8]

For document analysis, multiple iterations of *coding* were applied to a subset of the data to establish codes. Each code represents a category of related elements; e.g., ALGORITHMS (which includes the topics of *proofs*, *logic*, *runtime analysis*, and *optimization*). Figure 3 provides an overview of the coding process. The first three steps (1)–(3) served to *produce* the codes: First (in step 1), a randomly selected subset of the data (20%) was coded *inductively* by the first author (A1) to *establish* preliminary codes. Second (step 2), another randomly selected sample of the data (another 20%) was coded *deductively* to *validate* the preliminary codes. Third (step 3), the established codes were compared to the models of various literature reviews on the topic of computational competency models, refining the codes (Weintrop et al, 2016; Tikva and Tambouris, 2021; Murphy et al, 2016).

Hereafter, to validate (step 4), the first and second authors (A1+A2) collaboratively coded yet another random subset (20%). Then (in step 5), the second author (A2) coded an additional 20% alone. Simultaneously (in step 6), the first author coded the remaining 80% according to the refined codes and collaborative coding. Matching the codes individually identified by the first (A1) vs second author (A2), we obtain an IRR (Inter-Rater Reliability) agreement percentage of 84% and a Kappa of 0.79 which implies *substantial agreement* between the two coders (Gisev et al, 2013).

Naturally, course descriptions *can* be ambiguous or not paint the full picture of a course since a large part of the work (including decision-making) lies in the way it is taught. Consequently, the assumptions made here regarding courses may not equate fully to whether or not something *is* included in a given course. In Denmark, however, the vast majority of course exams are conducted in collaboration with an *external examiner* (aka, *censor*) from another university. This person is also charged

[8]Structure and documentation is still a work in progress. While some courses are in English, translation is still in progress for the remaining courses. The repository can be found here: https://github.com/sebastiannicolajsen/ipcrd

with ensuring the exam assesses the explicit learning goals stipulated in the course description. Also, any complaints are settled according to the course descriptions. Hence, course descriptions accurately account for the (intended) teaching.

5 Programmes with Programming? (RQ1)

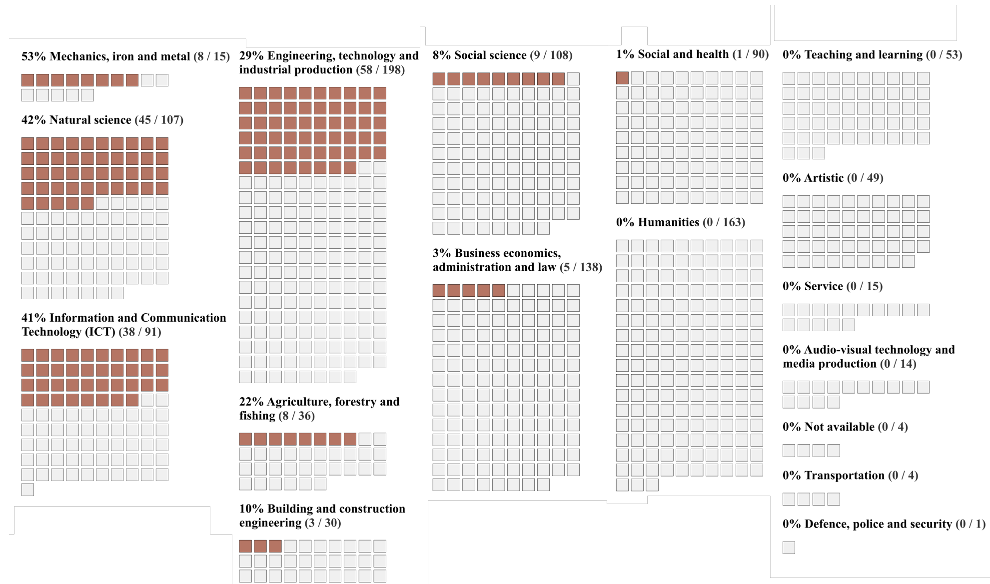

Fig. 4: Distribution of educational programmes *with* mandatory programming (red squares) vs educational programmes *without* mandatory programming (gray squares).

We examined a total of 679 different educational programmes. Since educational programmes are sometimes offered by varying institutions, this number corresponds to a total of 1,169 instances of educational programmes. Within those, we found that 175 educational programmes introduce mandatory programming through 130 courses, with some courses shared across multiple programmes. We thus observe that:

> **Observation 1a (Programmes):** Mandatory programming has emerged in **1 out of 6½** educational programmes (175 out of 1,169).

The size of this number, 1 in 6½ programmes (15%), is a testament to the massive effect of digitalization on the higher educational landscape. Computer programming is not just for computer science; mandatory programming has been deemed essential and emerged in many non-computing educational programs, which we explore below.

Figure 4 provides an overview of which educational *areas* include mandatory programming. For this overview, we used the official *governmental educational register* of Denmark, which *categorises* educational programmes according to their so-called *disciplinary domain*; e.g., *Natural Science* or *Humanities* (Danmarks Statistik, 2016).

However, since the register only provides a single category for each educational programme, some educational programmes will invariably be inappropriately categorized; e.g., *Health & Informatics* (at KU) is unilaterally categorized as *Information and Communication Technology (ICT)* and *Health & Welfare Technology* (at SDU) is categorized as *Engineering, technology, and industrial production*, despite both programmes indicating they are interdisciplinary. This is an unavoidable consequence of any uni-dimensional categorization. Even though the list is non-exhaustive and the categorisation does not account for interdisciplinary programmes, the overview does show that programming is not exclusively introduced within Computer Science (or the ICT domain). It shows that:

Observation 1b (Areas): Mandatory programming has emerged in all of the major disciplinary areas (with the exception of *Humanities*, *Teaching & Learning*, and *Artistic*).

Figure 5 shows how educational programmes featuring mandatory programming are distributed across institutions. We note that:

Observation 1c (Institutions): The vast majority (97%) of tertiary educational programmes where mandatory programming has emerged are at university (169 in 175).

It is, in fact, not surprising that colleges and other non-university institutions are less represented since they mostly offer other types of educational programmes; e.g., vocational education (or other types which we do not explore).

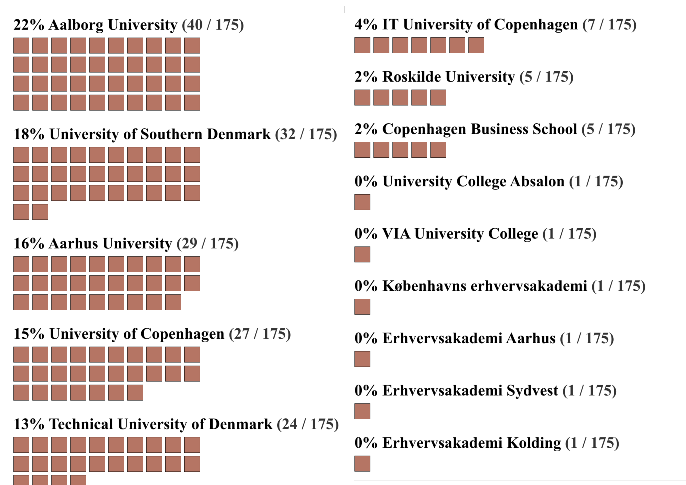

Fig. 5: Distribution of programmes with mandatory programming across institutions.

Figure 6 shows the educational programmes with mandatory programming according to their educational degree type. We see that:

Observation 1d (Degrees): Most educational programmes where mandatory programming has emerged (almost 2/3) are at the BSc level; a fifth (20%) are on professional bachelor & the BSc of engineering; only the last sixth (16%) are at the MSc level.

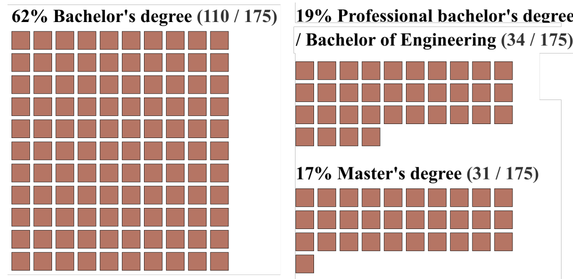

Fig. 6: Distribution of programmes across educational level.

Again, this pattern is expected since there is a large degree of coordination between bachelor's and master's degrees in Denmark. The master's degree is indeed often a *continuation* of a bachelor's degree; thus, if a master's degree expects programming competences, they will often have been introduced on the bachelor level.

In summary, many educational programmes include mandatory programming and cover a wide variety of disciplinary areas, institutions, and degree levels.

6 Role of Programming? (RQ2)

While the debate on whether or not to teach programming has been permeating the field of computing education research, an important question to ask first is *why* teach programming. While Ko et al (2020) and Evrard and Guzdial (2023) present a categorisation of computing for *expression*, *justice*, and *discovery*, we did not find reflections of these in isolation nor in relation to programming (Ko et al, 2020; Evrard and Guzdial, 2023). Instead, we identified that course curricula mention programming in two distinct ways: First, programming is described ***as a language*** (used to conduct activities *within* the domain of computing). Second, programming is described ***as a tool*** (used to conduct activities *outside* of computing). While this categorisation appears highly comparable to the perspective introduced by Denning et al (1989) of computing as *discipline-oriented thinking* or *tool use*, we use different terminology as they discuss computing broadly and we focus on programming. Further, as we will outline here, it is not two distinct categories but rather a spectrum of perspective. Figure 7 illustrates the number of *programmes* (as opposed to *courses*) we have identified which code to either of the two codes (or both), whereas white squares represent programmes we identified for which we did not code any of these codes. We will, throughout this paper, present the number of courses (out of 130 courses) that code to a particular topic and illustrate the distribution across disciplines using programmes (where 175 programmes share the 130 courses).

From Figure 7 it is apparent that despite the singular categorisation used by the national educational register, both types of descriptions (*as a language* & *as a tool*) appear across disciplines, except for *ICT* and *'Mechanics, Iron, and Metal'* (where the role of programming is exclusively *as a language*).

Taught *as a language* (to conduct activities *within* computing). 60% of courses (93 courses) explicit programming *as a language*, talking explicitly about a

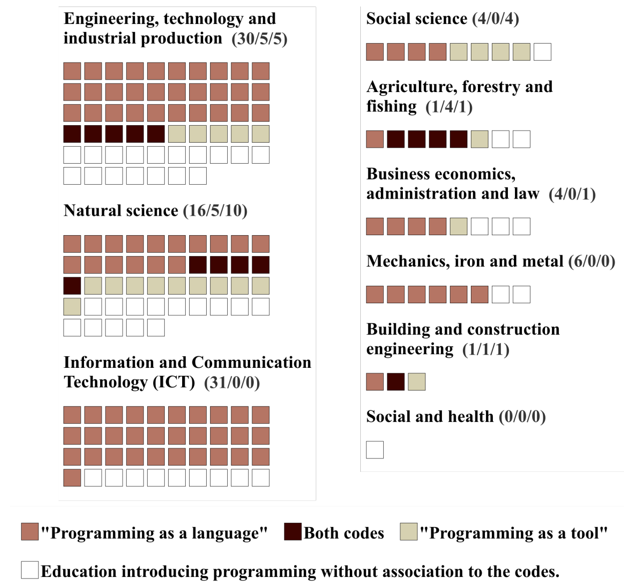

Fig. 7: Programmes coding to either language and/or tool.

particular programming language. While these courses intend to teach general-purpose programming, they might, like the course *Introduction to Programming*, have adjacent or complimentary goals, such as enabling communication with professional developers, educating vernacular/conversational programmers, incorporating ideas such as *computational thinking* (Shaw, 2022; Evrard and Guzdial, 2023):

> *The course provides students with a basic understanding of computational thinking and programming both for their own future use and for their ability to collaborate with experienced programmers and software developers. The students will learn about the concept of computational thinking and get a hands-on introduction to programming using the Python programming language. Programming and computational thinking are basic primitives in today's IT world.*
>
> — [Digital Design & Interactive Technologies, ITU]

Taught *as a tool* (used to conduct activities *outside* of computing). In contrast, 22% of courses (29 courses) explicitly describe programming to solve issues *outside* of computing or *as a tool* to achieve *something*; e.g., *"how to obtain, store, manipulate, and analyse data using relevant software and foundational machine learning approaches."* (*Economy*, AAU). Interestingly, programming languages are not only categorised as a *tool*, but as *software*, often connected to solving issues in a domain-specific context:

> *The course equips students with basic skills in data analysis which are required for analyzing socio-economic issues as well as learning more advanced methods in advanced semesters. In this regard, the course also* ***teaches the use of a statistical program, R.***
>
> — [Economy, AAU]

This is not exclusive to certain disciplines but appears in various ones, e.g., *Biology* (*Biomedicine*, SDU) or *Mathematics* (*Mathematics*, AAU). As argued in the description of the course *'Biostatistics in R 1,'* choosing a programming language as the primary software offers a broad range of specialized tools and individual tailoring:

> *Among currently available software suits, the R scripting language became very popular to deal with biostatistics and analysis of large data sets, as it (i) provides a vast number of statistical tools, (ii) allows adaptation of the analysis to any experimental design, (iii) offers simple commands to operate on entire data sets, (iv) provides a wide range of methods for data visualization and (v) has a large and active community of researchers developing new tools.*
>
> — [Biochemistry & Molecular Biology, SDU]

Taught *as a language & as a tool.* Interestingly, we found that 6% of courses (8 courses) mention programming both *as a language* and also *as a tool.* It is consequently important to think of these categorisations as a *spectrum* rather than a *dichotomy.* While existing research suggests a more strict categorisation of programming, we argue that the diversity of disciplines requires a "looser definition" of programming's purpose, namely that it can be: a *language*, a *tool*, or *both* (*a language & a tool*):

> **Observation 2 (Role):** In educational programmes where mandatory programming has emerged, programming has manifested itself on the spectrum between being used ***as a language*** (for computing-specific activities) and ***as a tool*** (for activities in other domains) across disciplines.

7 Delineation of Programming? (RQ3)

The *role* of programming manifested itself on a spectrum from being used *as a language* and *as a tool.* This may largely relate to whether programming is the *sole* content of the course, which, in turn, affects the design and curriculum of a course. This is one of the reasons why the debate on *how* to integrate computing is a fundamental choice (Caspersen et al, 2017; Palumbo, 1990). Based on the curricula analysed, we were able to ascertain how this was done for 123 of the courses. Some courses are taught ***in isolation***; i.e., they introduce programming in a standalone course. Other courses are taught ***in combination***; i.e., they combine programming with other topics. Figure 8 illustrates the distribution of courses taught *in isolation*/*in combination* across the identified programmes. Again, we see that no disciplinary area is dominated by either approach to introducing programming (*in isolation* or *in combination*).

Taught *in isolation.* 60% of courses (79 out of the 130 courses) appear to focus *exclusively* on programming. Despite this, the courses vary largely in their focus. Some courses highlight common language features such as numbers, strings, branching, logical operators, and functions (e.g., *Chemistry & Technology*, DTU) or object-oriented programming (e.g., *Business Administration & Digital Business*, CBS). Others focus on computer architecture, algorithms, and data structures (e.g., *Computer Technology*, AAU) or compilers, linkers, and pointers (e.g., *Electrical Energy Technology*, AU). Some courses mix multiple of these areas (e.g., *Software Design*, ITU). While no single area appears more dominant, they all focus solely on programming. Despite the

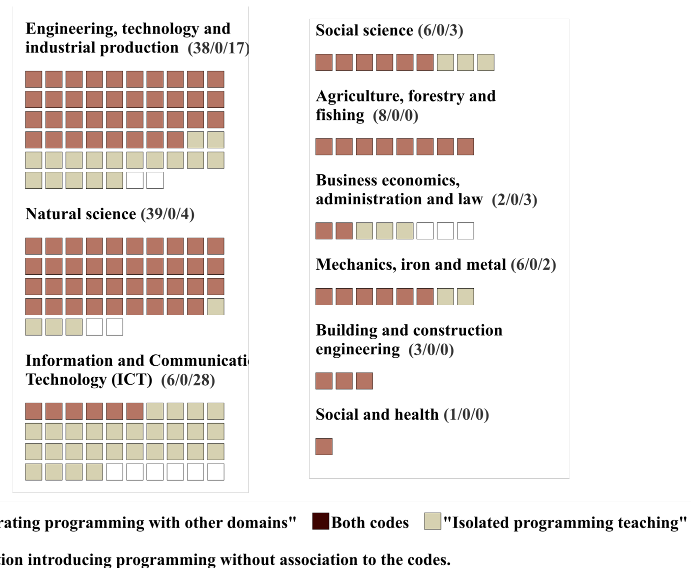

Fig. 8: Programmes containing courses taught *in isolation* (standalone programming courses) vs courses taught *in combination* (programming combined with other topics).

standalone nature of the course, courses like *Programming 1* argue for the importance of programming in itself:

> *In today's healthcare, technology is included to a greater and greater extent. These technologies are largely based on a combination of electronics and programs. Therefore, it is necessary for health technology engineers to obtain a certain routine in reading and writing programs. In this course great emphasis is put on practical exercises to achieve this routine.*
> — [Health Technologies, AU]

Taught *in combination.* In contrast, 33% of courses (44 courses) *combine* the introduction of programming *with other disciplines*; e.g., the analysis and synthesis of geodata (*Geography and Geoinformatics*, KU). In particular, such courses often focus on *applying* programming (following Observation 2 regarding programming *as a tool*, *outside* of computing):

> *The purpose of the course is to give students basic insight into data science [Datavidenskab]. This is done with a special focus on data handling, exploration and visualization. The objective is to provide a broad overview of data science and introduce the students to the basic tools and methods for handling the typical work routines encountered in a project [...]*
> — [Data Science, AU]

While these observations do not solve the question of how to delineate computing in general education, they highlight that both approaches (*in isolation* & *in combination*) appear to work in the context of tertiary education and, further, that both approaches are applied across disciplinary areas:

Observation 3 (Delineation): Across disciplines, mandatory programming has emerged as taught both ***in isolation*** as well as ***in combination*** (with other topics).

8 Kind of Programming Competences? (RQ4)

Given that programming is taught both ***in isolation*** and ***in combination*** (with other topics), it is important to consider what kind of competences a course aims to establish. For RQ4 (*"What kind of competences appear across disciplines?"*), we found three different *"families of (related) competences"*:

- (1) *Problems & Problem-solving*;
- (2) *Collaboration (incl. Communication)*; and
- (3) *Abstraction & Algorithms.*

We now consider each of these *competence families*, in turn.

8.1 Problems & Problem-solving

Problems. The literature suggests that solving problems (aka, *problem-solving*) is a primary function of computing (Gries, 1974; Kalelioglu et al, 2016). When introducing problem-solving outside of computing for, say, *biologists*, it would be highly relevant to situate these problems in the context of the domain: *biology.* This essentially means integrating computing *into* the domain and combining it with the topics of the domain.

Figure 9 illustrates how *domain problems* appear across disciplines. According to our study, 19% of courses (25 out of the 130 courses) have explicit domain problems in their curricula. We see *domain problems* appearing in all domains, except in the context of ICT, where problem-solving is presumably independent of any particular domain:

Observation 4a (Domain Problems): Mandatory programming has emerged with a focus on *domain problems* across all disciplines (except for the ICT disciplinary area).

Problem-solving. Like existing literature, some courses highlight the central function of problem-solving; for instance:

> *Programming is about solving problems: Programs that can be executed on a computer are developed to solve a problem. Assuming that we understand the problem to be solved, we face two challenges. The first is how to instruct the computer and the second is how to judge whether the problem has been adequately solved [...].*
>
> — [Computer Technology, AU]

The constituents of problem-solving, specifically for (object-oriented) programming, typically include: (1) *problem identification and analysis*, (2) *design*, (3) *implementation*, and (4) *validation* (Madsen et al, 1993).

We did not find equally clear explications of the entire problem-solving process in the course descriptions. Rather, curricula focused on *design*, *implementation* (or both), and *validation.* Figure 10 illustrates the distribution of these processes onto programmes across disciplines. 40% of courses (52 courses) mention both *design* and *implementation*; 33% of courses (43 courses) exclusively mention *implementation*; and 3% of courses (4 courses) exclusively mention *design* (omitted from the Figure due low

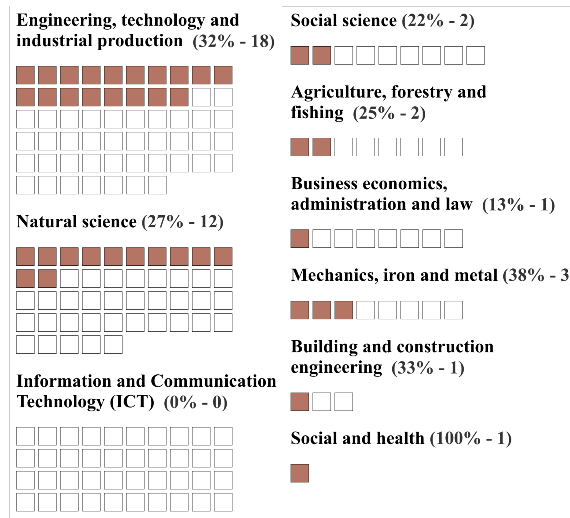

Fig. 9: Programmes with mandatory programming courses that explicitly mention *domain problems*.

numbers). Further, 40% of courses (52 courses) mention the teaching of validation; e.g., testing and/or debugging. Again, Figure 10 shows that these steps of problem-solving are widely recognized in curricula across disciplines, but also that *implementation* (in isolation, without *design*) is widely practised. It should be noted that validation was only registered to be present when either both design and implementation were described *or* implementation exclusively.

While all these courses focus on *design* and/or *implementation*, they vary in terms of what this implies; e.g., *"understanding methods for program design and be able to distinguish between good and bad praxis"* (*Digitalization & Application Development*, AAU), *"conduct analysis of a specification to identify phenomena"* (*Software Engineering*, SDU), and *"identify requirements of a program, prepare a technical design, implement and test it"* (*IT, Communication, & Organisation*, AU). Similarly, courses vary in terms of what validation implies, e.g., planning and executing tests (*Mathematics & Economy*, SDU), reason informally about programs and relating it to test cases (*Electro Technology*, AU), and understanding the role of debugging (*Software Engineering*, SDU). From this, we observe the following:

> **Observation 4b (Problem-solving):** A subset of problem-solving activities are widely emphasized across disciplines; i.e., *design+implementation*, or *implementation* in isolation, and *validation*.

In relation to problem-solving, and most often in combination with *validation*, we identified that 22% of courses (29 courses) *also* highlight the ability to read/write documentation of programs. While documentation is not explicitly part of the identified problem-solving steps, it appears across disciplines, as shown in Figure 11.

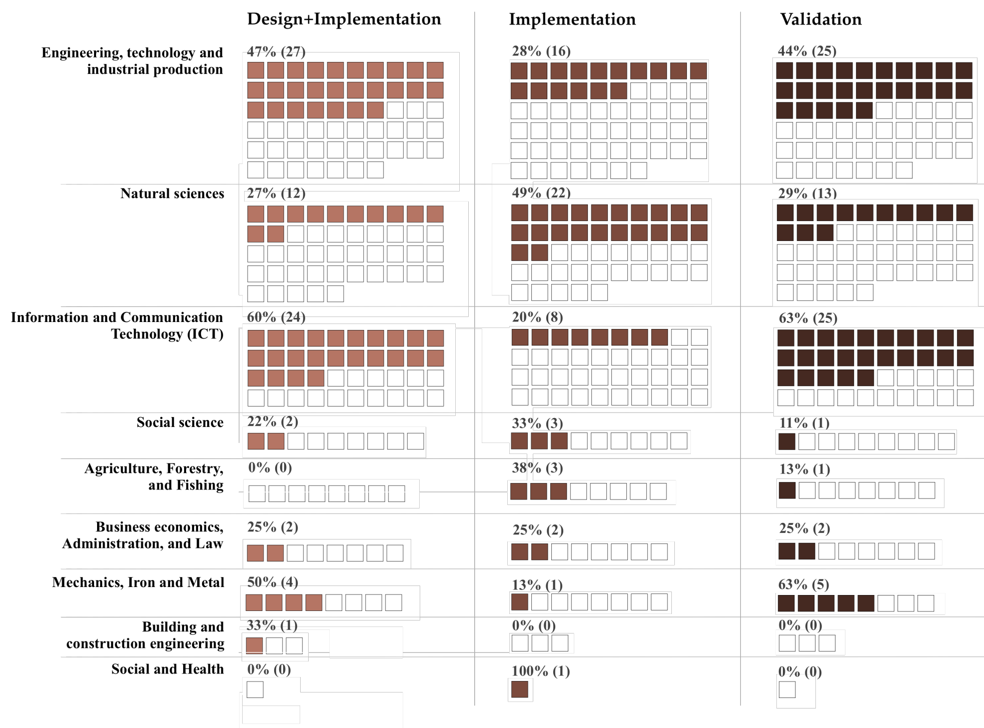

Fig. 10: Programmes describing *design+implementation*, *implementation* in isolation, and *validation*

Although documentation is not as widespread as the existing steps of problem-solving, we observe the following:

> **Observation 4c (Documentation):** A subset of courses introducing steps of problem-solving, particularly validation, also highlight the ability to read/write documentation.

8.2 Collaboration (incl. Communication)

Previous research highlights the need for conversational or vernacular programmers to be able to discuss, work, communicate and collaborate with professional developers (Shaw, 2022; Evrard and Guzdial, 2023). Courses largely focus on collaborative competences, i.e., some kind of collaboration is mentioned in 34% of courses (44 of the 130 courses). These competences span the various programmes as seen in Figure 12. Some courses, for instance, focus on *"working in groups"* (*Software Technology*, DTU), *"handle development-oriented situations"* (*Computational Medicine*, SDU), or *"reviewing others' solutions"* (*Electronics*, AU) while others argue for *"readable code"* (*Chemistry & Technology*, DTU) and the *"ability to express decisions made"* (*Electro Technology*, DTU).

Further, 11% of courses (15 courses) focus specifically on the ability to communicate with various stakeholders, e.g.:

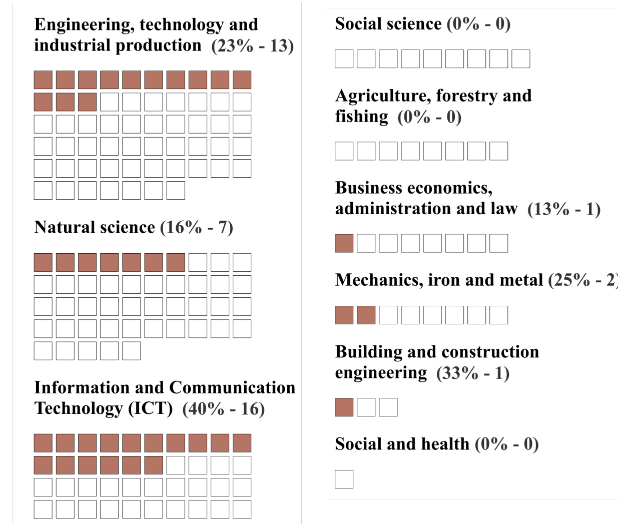

Fig. 11: Programmes introducing the ability to read/write documentation.

> *Communicate the conclusions of statistical analysis clearly and effectively; i.e., identify connections between basic statistics and the real world.*
>
> — [*Market & Culture Analysis*, CBS]

In conclusion, we make the following observation:

> **Observation 5 (Collaboration):** Collaboration (including communication) competences appear across disciplines.

8.3 Abstraction & Algorithms

Abstraction. While literature highlights abstraction as a core competence within computational thinking (Kalelioglu et al, 2016; Haseski et al, 2018), it only appears in 9% of courses (12 courses of the 130 courses). This is despite extending our coding to include related concepts, such as decomposition. Therefore, we do not visualize the distribution across educational programmes, but merely observe the following:

> **Observation 6a (Abstraction):** *Abstraction* is almost not apparent in course curricula (despite being hailed, in the literature, as *'crucial for programming'*).

Algorithms (or *algorithmic thinking*) is another often mentioned competence of computational thinking (Haseski et al, 2018; Knuth, 1985); it appears in 29% of courses (38 courses). Figure 13 illustrates the disciplines wherein algorithms appear. Now, algorithms are much less apparent in natural sciences and ICT than one may expect. This may be caused by the fact that algorithms are usually not taught on *introductory* programming courses but only later in the programme for technical (and programming) specific educational programmes.

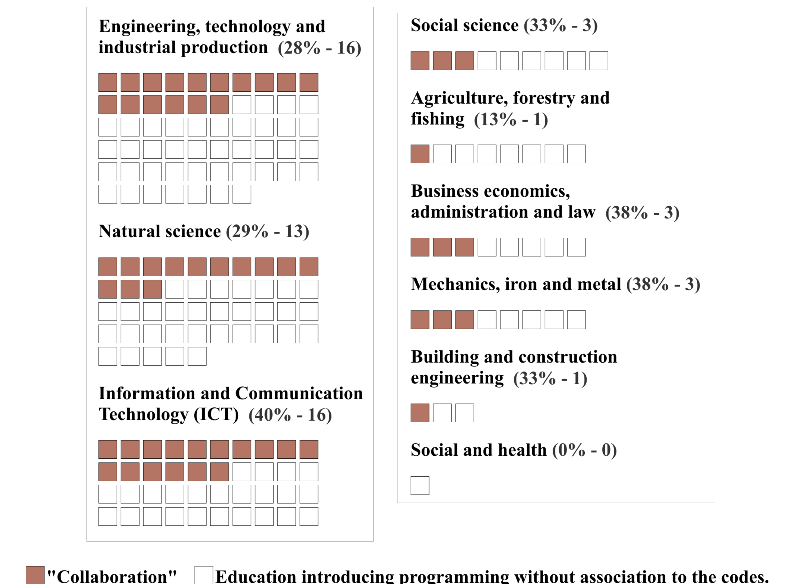

Fig. 12: Programmes describing collaborative skills.

These courses vary largely on *what* is implied in relation to algorithms, whether this be the introduction of algorithms and algorithmic techniques (e.g., *Data Science*, ITU), optimization (e.g., *Bioinformatics & Systems Biology*, DTU), or understanding of run-time complexity and their uses (e.g., *Bio Informatics*, AU). While the majority of these courses discuss algorithms, a few argue for the use of algorithms in their respective disciplines:

> *The participants will after the course have detailed knowledge of the concept of computational thinking and programming in a bioinformatics context, and have acquired practical experience in analyzing and solving computational problems using algorithmic and machine learning techniques in a bioinformatics context.*
>
> — [Bioinformatics, AU]

This is further supported by other courses, e.g., *Programming & Data Processing* (*Digital Innovation & Management*, ITU) where the argument is the importance to future employers. In conclusion, we make the following observation:

> **Observation 6b (Algorithms):** *Algorithms* are apparent across disciplines, however, not to the same extent as other competences.

9 Limitations

We now consider the limitations of our study.

Curriculum as the subject of analysis? Similar to Becker and Fitzpatrick (2019), we recognize that our analysis is more about concepts than the depth to which they are covered during teaching/learning activities within courses. However, we do

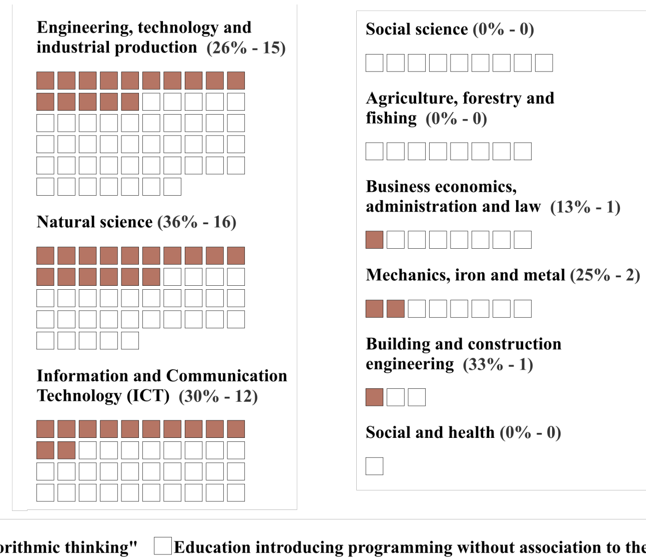

Fig. 13: Programmes including algorithms.

retain our position that these yield arguments of courses' existence and bring forth concepts which are *central* to the course (Winsløw, 2006). Further, Becker & Fitzpatrick argue that learning outcomes are a direct mechanism to gauge expectations of students, however, we decided to include everything accessible in course descriptions to paint the most detailed picture possible. Naturally, elements thereof will not be direct reflections of what we expect of students, yet due to the nature of some institutions following EQF, this would have yielded less representative results.

Incomplete data collection? The data was harvested and aggregated by a single researcher using the public education guide.[9] Due to the structure of the guide and the use of a single researcher, there is a risk of inadvertently excluding courses, thereby educational programmes introducing programming. However, we argue that the variety of programmes (according to their disciplinary area) we have identified suggests we have covered a significant portion of different educational programmes.

Inappropriate categorisation of programmes? As previously stated, the categorisation provided by the governmental educational register is rather limited in its recognition of interdisciplinary programmes (Danmarks Statistik, 2016). This limits our ability to provide a factual diagrammatic overview of programming course distribution across disciplines. Despite this, our observations appear to manifest across all the disciplines in which we identified programming courses. We argue that these tendencies exist across disciplines, even when applying a strict, unilateral categorisation of disciplines.

Analysis process? While we took an inductive approach to establishing codes, we also compared our findings to several frameworks of computational thinking. It is important to state that this does not entail adopting all content from existing

[9] The Education Guide: https://ug.dk

frameworks or eliminating elements which do not correspond to those frameworks. We, therefore, found elements (such as collaboration) which is not explicit in such frameworks. However, comparing to such frameworks still poses a threat of biasing our findings.

Generalization of context? Naturally, our findings are tied to the context of Denmark. However, given that Denmark adheres to European guidelines and frameworks, one may expect the findings to generalize to other European countries. Presumably, our case parallels countries where programming emerges organically through a bottom-up process at autonomous universities with the freedom to design their own educational programmes.

What about generative artificial intelligence? Generative AI is already impacting the global educational landscape, and educational institutions are already trying to figure out how to harness its benefits and mitigate its drawbacks. The jury is still out on predicting the impact on education, in general, and on programming, in particular. In any case, our study provides a "snapshot" of the state of teaching programming in higher education before it is likely disrupted by the era of generative AI.

10 Conclusion

In conclusion, we have explored 1,169 higher educational programmes and their course descriptions (from 2023). We found that 175 of the educational programmes (sharing 130 courses) include mandatory courses introducing programming. These are programmes which themselves manage their curricula, and, therefore, are expressions of individual needs for their disciplinary area. Using these course descriptions, we have explored the inclusion of programming (and computing) in a wide range of disciplinary areas. Based on our findings, we here provide a series of recommendations for educators. We recognize that these recommendations deserve further investigation to ensure consideration of the lived realities of operating in diverse universities. Thus, we present these recommendations for individual educators and policymakers to reflect upon, but also to encourage further exploration of them individually. First, we recommend that:

> **Recommendation 1a:** Any educator or policymaker should consider the inclusion of programming/computing given its widespread inclusion in higher education already.

Specifically, through **Observation 1(a–d)** used in addressing **what kind of educational programmes introduce programming (RQ1)**, we found that mandatory programming has emerged in 1 out of 6½ educational programmes. Further, mandatory programming appears within the majority of the (large) disciplinary areas according to the categorisation of the *governmental educational register*, with some exceptions (e.g., *Humanities* and *Teaching & Learning*), potentially due to the unilateral categorisation applied (Danmarks Statistik, 2016). Thus, it is apparent that programming is mandated for a large, diverse group of educational programmes. A decision made by the institutions and programmes *themselves*. Given this widespread inclusion in a rather conservative analysis of courses/programmes, we argue, like Haigh (1985), that educators and policymakers should consider the inclusion of programming/computing. In extension hereof, we recommend that:

Recommendation 1b: Educators need to take ownership of *what* introducing programming entails and *how* it should be presented.

Based on the widespread inclusion seen through **Observation 1a–d**, it should be clear that programming is not a possession of computer science but something that belongs to all disciplines. Historically, this has always been the case. Already in the 1950s, computing was rooted in electrical engineering, natural sciences, and mathematics; its pioneers sought to *design*, *construct*, and *utilize* large-scale (physical) machine computing (Tedre et al, 2018). As the community grew in the 1960s, so did the need for a unique identity, spawning ideas such as 'Algorithmic Thinking' (Tedre and Denning, 2016). However, programming long haunted the image of computing, regarded as an esoteric craft that raised questions about whether users could be educated in their use (Tedre et al, 2018). To this end, many tried to elevate programming into an esteemed academic discipline, which led to the introduction of computer science (Tedre et al, 2018). However, at the same time, the increased use of computers led to various other computational disciplines (Denning and Tedre, 2021). Consequently, an ACM task force developed their report *"Computing as a Discipline,"* changing the narrative from the field of *'Computer Science'* to the field of *'Computing,'* acknowledging the widespread use of computing (Tedre et al, 2018). Therefore, we do not believe programming or computing to be a possession of computer science; rather, this is a remnant of the past—it has been and always will be a significant part of other disciplines. As such, programming is something relevant to all of higher education.

We recognise including computing/programming is a daunting task, further complicated by the fact that the majority of research on introductory programming exists within the domain of computer science (e.g., Becker and Fitzpatrick (2019) and Luxton-Reilly et al (2018)). We thus recommend that:

Recommendation 2: Educators should consider what programming can do for *their* discipline.

To support educators in doing so, following **Observation 2**, we propose educators view **the role of programming (RQ2)** as a spectrum from being *as a language* (to solve computing problems) or *as a tool* (to solve problems outside of the domain of computing). While existing research suggests a more stringent categorisation of computing's potential purposes, we believe this only limits the opportunities to be found in individual disciplines (Ko et al, 2020; Evrard and Guzdial, 2023). Further, we found no explicit examples of the identified categories in the explored programmes. By instead recognising it as a spectrum and focusing on how programming can support a given domain (by existing somewhere on this spectrum), tailoring and motivating its use becomes simpler compared to appropriating a certain category. This is further reinforced by research arguing computing education should; be tailored to its audience to increase interest (Chakrabarty and Martin, 2018; Ott et al, 2018); introduce experiences which meet students' career goals (Thiry et al, 2011), and; support diversity in types of programmers (vernacular and professional) (Shaw, 2022; Evrard and Guzdial, 2023).

When educators have identified what programming can do for *their* discipline, they are left with challenges such as how to include programming into their programme

(i.e., *in isolation* or *in combination* with another area) and which competences to teach. These challenges are largely intertwined, and we recommend that:

> **Recommendation 3:** The purpose and competences required should determine the delineation of introductory programming.

In regards to the question of how to **delineate programming (RQ3)**, through **Observation 3**, we found that mandatory programming is being introduced both through standalone courses (*in isolation*) and by integration with other areas (*in combination*). This difference will obviously affect the contents of such courses. Standalone introductory programming courses (*in isolation*) likely introduce different elements; e.g., language features, object-oriented programming, and/or algorithms and data structures. Courses introducing programming *in combination* with other areas, on the other hand, may involve analysis and synthesis of data or solving problems. We therefore conclude that both approaches appear feasible in all disciplines. While the question of how to delineate programming is still being debated, we believe it to be more crucial to consider the *purpose* and *contents* of the course in relation to the *purpose* and *contents* of the programme (Caspersen et al, 2017; Palumbo, 1990). By focusing on these constituents, the question of delineation will become apparent based on the choices made. For example, if the purpose of programming is to solve problems computationally within a domain, then the choice will likely be to integrate programming into a course. If the purpose is instead to provide a strong fundamental understanding of programming as a language, then an isolated course may be preferred. Thus, the choice of delineation should be determined by the necessary competences, for which we recommend that:

> **Recommendation 4:** Educators should include *problem-solving* and *collaboration* competences in their introductory programming course no matter the discipline.

Through **Observations 4–6** we see that the most important **kinds of programming competences (RQ4)** across disciplines appear to be *problem-solving*; in particular, *design*, *implementation*, and *validation*. Interestingly, this diverges from existing research, which also suggests high importance of both abstraction and algorithms (Lu et al, 2022; Becker and Fitzpatrick, 2019). While these competences did appear, they were far less apparent across disciplines than, e.g., *problem-solving*. The importance of abstraction and algorithms may result from existing research focusing on programming in a computer science context. However, we argue that educators introducing programming in diverse disciplines should, in practice, *not* focus on these. Instead, any discipline aiming to allow graduates to solve problems computationally, *problem-solving* ought to be a central competence. Moreover, educators should consider what problem-solving implies in *their* field. For example, Becker (2023) validly criticises computer science for not critically examining the plurality and subjectivity of problems, i.e., that computer science views problems as singular objective truths.

Further, educators should consider *collaboration* competences, which also appear across disciplines. With the continuously rising application of computing, collaboration ought to be crucial to support the construction of software supporting various domains. This is also supported by vernacular programmers' synonymous name, '*conversational*' programmers (Evrard and Guzdial, 2023). In addition, one can also argue

that problem-solving and collaborative competence are closely connected in student work and professional endeavours (Raj et al, 2021).

While we here consider these three competences in relation to programming, educators may find that they are better taught *without* programming. Here, we simply shed light on the diverse inclusion of these competences in programming courses. After all, as stated by Evrard and Guzdial (2023), programming is a *means* to learn computing.

While laudable research explores programming in the context of computer science, we believe this study to be the first explicit step towards exploring computing/programming curricula in a broad context. By conducting future research and building on this, we may identify and improve the application of computing in individual disciplines, supporting educators and policymakers in following the above recommendations. Concretely, this should entail both exploration of different curricula in diverse contexts and assessment of different pedagogical approaches, as is already being conducted in the domain of computer science. Further, we hope to inspire similar work *outside* of Denmark to understand international similarities and differences. Last, while the profiliation of generative AI is still underway, we believe it may support educators in strengthening competences associated with programming, in particular, *problem-solving*.

Acknowledgement

The authors would like to thank Samantha Breslin and Marisa Cohn for their impeccable attention to detail and feedback during the study's design and the development of our analysis.

References

Aleksić V, Ivanović M (2016) Introductory programming subject in european higher education. Informatics in Education 15(2):163–182

Ardito C, Buono P, Costabile MF, et al (2012) End users as co-designers of their own tools and products. Journal of Visual Languages & Computing 23(2):78–90

Barendsen E, Steenvoorden T (2016) Analyzing conceptual content of international informatics curricula for secondary education. In: Informatics in Schools: Improvement of Informatics Knowledge and Perception: 9th International Conference on Informatics in Schools: Situation, Evolution, and Perspectives, ISSEP 2016, Münster, Germany, October 13-15, 2016, Proceedings 9, Springer, pp 14–27

Becker BA, Fitzpatrick T (2019) What do cs1 syllabi reveal about our expectations of introductory programming students? In: Proceedings of the 50th ACM technical symposium on computer science education, pp 1011–1017

Becker C (2023) Insolvent: How to reorient computing for just sustainability. MIT Press

Bennedsen J, Caspersen ME (2006) Abstraction ability as an indicator of success for learning object-oriented programming? ACM Sigcse Bulletin 38(2):39–43

Biggs J, Tang C, Kennedy G (2022) Teaching for quality learning at university 5e. McGraw-hill education (UK)

Biggs JB, Collis KF (2014) Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome). Academic Press

Brabrand C, Dahl B (2008) Constructive alignment and the solo taxonomy: a comparative study of university competences in computer science vs. mathematics. In: Conferences in Research and Practice in Information Technology, Australian Computer Society, pp 3–17

Brabrand C, Dahl B (2009) Using the solo taxonomy to analyze competence progression of university science curricula. Higher Education 58:531–549

Bundsgaard J, Bindslev S, Caeli EN, et al (2019) Danske elevers teknologiforståelse: resultater fra ICILS-undersøgelsen 2018. Aarhus Universitetsforlag

Caspersen ME, Iversen OS, Nielsen M, et al (2017) Computational thinking. Dolin, G Holten Ingerslev & Hanne Sparholt Jørgensen (Red), Gymnasiepædagogik En grundbog København: Hans Reitzels pp 470–478

Chakrabarty S, Martin F (2018) Role of prior experience on student performance in the introductory undergraduate cs course. In: Proceedings of the 49th ACM Technical Symposium on Computer Science Education, pp 1075–1075

Christensen IM, Marcher MH, Grabarczyk P, et al (2021) Computing educational activities involving people rather than things appeal more to women (recruitment perspective). In: Proceedings of the 17th ACM Conference on International Computing Education Research, pp 127–144

Crick T, Sentance S (2011) Computing at school: stimulating computing education in the uk. In: Proceedings of the 11th Koli Calling International Conference on Computing Education Research, pp 122–123

Curzon P, McOwan PW (2017) The power of computational thinking: Games, magic and puzzles to help you become a computational thinker. World Scientific

Danmarks Statistik (2016) Uddannelsesregister. URL https://www.dst.dk/Site/Dst/SingleFiles/GetArchiveFile.aspx?fi=uddannelse&fo=vejluddreg--pdf&ext=%7B2%7D, accessed on 21 05, 2024

De Raadt M, Watson R, Toleman M (2002) Language trends in introductory programming courses. Working paper

Denning PJ, Tedre M (2021) Computational thinking: A disciplinary perspective. Informatics in Education 20(3):361

Denning PJ, Comer DE, Gries D, et al (1989) Computing as a discipline. Computer 22(2):63–70

DiSessa AA (2000) Changing minds: Computers, learning, and literacy. Mit Press

Evrard G, Guzdial M (2023) Identifying the computing education needs of liberal arts & sciences students. In: Proceedings of the 23rd Koli Calling International Conference on Computing Education Research (Koli Calling '23)

Gal-Ezer J, Stephenson C (2014) A tale of two countries: Successes and challenges in k-12 computer science education in israel and the united states. ACM Transactions on Computing Education (TOCE) 14(2):1–18

Gisev N, Bell JS, Chen TF (2013) Interrater agreement and interrater reliability: key concepts, approaches, and applications. Research in Social and Administrative Pharmacy 9(3):330–338

Gretter S, Yadav A (2016) Computational thinking and media & information literacy: An integrated approach to teaching twenty-first century skills. TechTrends 60:510–516

Gries D (1974) What should we teach in an introductory programming course? In: Proceedings of the fourth SIGCSE technical symposium on Computer science education, pp 81–89

Haigh RW (1985) Planning for computer literacy. The Journal of Higher Education 56(2):161–171

Haseski Hİ, İlic U, Tuğtekin U (2018) Defining a new 21st century skill-computational thinking: Concepts and trends. International Education Studies

Ilomäki L, Kantosalo A, Lakkala M (2011) What is digital competence? Linked portal

Jones SP, Mitchell B, Humphreys S (2013) Computing at school in the uk. CACM Report

Kalelioglu F, Gulbahar Y, Kukul V (2016) A framework for computational thinking based on a systematic research review. Baltic Journal of Modern Computing

Knuth DE (1985) Algorithmic thinking and mathematical thinking. The American Mathematical Monthly 92(3):170–181

Ko AJ, Oleson A, Ryan N, et al (2020) It is time for more critical cs education. Communications of the ACM 63(11):31–33

Lu C, Macdonald R, Odell B, et al (2022) A scoping review of computational thinking assessments in higher education. Journal of Computing in Higher Education 34(2):416–461

Luxton-Reilly A, Simon, Albluwi I, et al (2018) Introductory programming: a systematic literature review. In: Proceedings companion of the 23rd annual ACM conference on innovation and technology in computer science education, pp 55–106

Madsen OL, Møller-Pedersen B, Nygaard K (1993) Object-oriented programming in the BETA programming language. Addison-Wesley

Mason R, Cooper G (2014) Introductory programming courses in australia and new zealand in 2013-trends and reasons. In: Proceedings of the Sixteenth Australasian Computing Education Conference-Volume 148, pp 139–147

Mason R, Simon, Becker BA, et al (2024) A global survey of introductory programming courses. In: Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1, pp 799–805

Méhaut P, Winch C (2012) The european qualification framework: skills, competences or knowledge? European educational research journal 11(3):369–381

Menekse M (2015) Computer science teacher professional development in the united states: a review of studies published between 2004 and 2014. Computer Science Education 25(4):325–350

Murphy E, Crick T, Davenport JH (2016) An analysis of introductory programming courses at uk universities. arXiv preprint arXiv:160906622

Nicolajsen SM, Pischetola M, Grabarczyk P, et al (2021) Three+ 1 perspectives on computational thinking. In: Proceedings of the 21st Koli Calling International Conference on Computing Education Research, pp 1–11

Ott L, Bettin B, Ureel L (2018) The impact of placement in introductory computer science courses on student persistence in a computing major. In: Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education, pp 296–301

Palumbo DB (1990) Programming language/problem-solving research: A review of relevant issues. Review of educational research 60(1):65–89

Pears A, Seidman S, Malmi L, et al (2007) A survey of literature on the teaching of introductory programming. Working group reports on ITiCSE on Innovation and technology in computer science education pp 204–223

Raj R, Sabin M, Impagliazzo J, et al (2021) Professional competencies in computing education: pedagogies and assessment. In: Proceedings of the 2021 Working Group

Reports on Innovation and Technology in Computer Science Education. p 133–161

Robins A, Rountree J, Rountree N (2003) Learning and teaching programming: A review and discussion. Computer science education 13(2):137–172

Shaw M (2022) Myths and mythconceptions: What does it mean to be a programming language, anyhow? Proceedings of the ACM on Programming Languages 4(HOPL):1–44

Statistik D (2022) Ungdomsuddannelser - uddannelsesaktiviteter på gymnasiale uddannelser. URL https://www.dst.dk/da/Statistik/emner/uddannelse-og-forskning/fuldtidsuddannelser/ungdomsuddannelser

Stephens S, O'Donnell D, McCusker P (2007) Computing careers and irish higher education: A labour market anomaly. Industry and Higher Education 21(2):159–168

Tedre M, Denning PJ (2016) The long quest for computational thinking. In: Proceedings of the 16th Koli Calling international conference on computing education research, pp 120–129

Tedre M, Simon, Malmi L (2018) Changing aims of computing education: A historical survey. Computer Science Education 28(2):158–186

Thiry H, Laursen SL, Hunter AB (2011) What experiences help students become scientists? a comparative study of research and other sources of personal and professional gains for stem undergraduates. The Journal of Higher Education 82(4):357–388

Tikva C, Tambouris E (2021) Mapping computational thinking through programming in k-12 education: A conceptual model based on a systematic literature review. Computers & Education 162:104083

Valentine DW (2004) Cs educational research: a meta-analysis of sigcse technical symposium proceedings. ACM SIGCSE Bulletin 36(1):255–259

Weintrop D, Beheshti E, Horn M, et al (2016) Defining computational thinking for mathematics and science classrooms. Journal of science education and technology 25:127–147

Wing JM (2006) Computational thinking. Communications of the ACM 49(3):33–35

Winsløw C (2006) Didaktiske elementer. En indføring i matematikkens og naturfagenes didaktik 1:229–242

Appendix

#	Grade definition	ECTS
12	For an excellent performance that completely meets the course objectives, with ***no or only a few insignificant weaknesses***.	A
10	For a very good performance that meets the course objectives, with ***only minor weaknesses***.	B
7	For a good performance that meets the course objectives but also displays ***some weaknesses***.	C
4	For a fair performance that adequately meets the course objectives but also displays ***several major weaknesses***.	D
02	For a sufficient performance that ***barely meets the course objectives***.	E
00	For an insufficient performance that ***doesn't meet the course objectives***.	Fx
-3	For a performance that is ***unacceptable in all respects***.	F

Fig. 14: The Danish grade scale for which individual grades are defined as *the degree of fulfilment of explicit learning goals* (aka, *course objectives*). The five grades above the single line are *pass* grades; the two grades below the line are both *fail* grades. (The last column, to the far right, gives the ECTS-equivalent conversion grades for comparison within the European Union.) Translated from Danish (Brabrand and Dahl, 2009).