Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Jens Mæland
Jonas Sæther

Automated generation of variant programming tasks using large language models

Master's thesis in Computer Science
Supervisor: Guttorm Sindre
June 2023

NTNU
Norwegian University of
Science and Technology

Jens Mæland
Jonas Sæther

Automated generation of variant programming tasks using large language models

Master’s thesis in Computer Science
Supervisor: Guttorm Sindre
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Abstract

Programming as a discipline has seen an increase in popularity ever since its inception. More and more students enroll in introductory programming courses, leading to an increase in resources needed by faculties to ensure effective education. Following the increase in student mass, lecturers find themselves spending more of their available time grading and creating exercises for students. One way of making introductory CS courses more efficient is to automatically generate variant tasks for use in both coursework and exams.

The aim of this thesis was to design and create a prototype for programming variant task generation meant for digital programming coursework. The thesis also evaluates the effect of this prototype, in regards to perceived usability by professors and other teaching staff, and students input regarding the generated variant tasks. The system makes use of large language models from OpenAI's GPT line, and seeks to increase efficiency in courses by automating task creation on a large scale, and provide teaching staff with creativity powered by large language models.

The thesis is rooted in a pre-master project, in which a prototype not powered by AI was built to generate programming task variants. After developing said prototype and examining the flaws of such a system, the project pivoted to make use of recent language model improvements in the hopes of providing better results. The thesis considers several task types, and several ways of generating meaningful tasks. Through iterative development and end user testing, an assessment of the system deemed it to have some advantages over manually creating programming tasks, and to be an effective aid in creating programming tasks to further expand upon. There are several shortcomings of the system, but as a prototype it shows promise and outlines a future where large language models can be used to reliably generate creative variant tasks that fosters learning.

Sammendrag

Programmering har utelukkende økt i populartitet siden det oppstod. Flere og flere studenter melder seg på introduksjonskurs i programmering, noe som fører til en dramatisk økning i ressursbehovet fra fakulter for å sikre god utdanning. Den økte studentmassen drar med seg økte krav til forelesere, som må bruke mer av sin tilgjengelige tid på å evaluere innleveringer og lage øvinger, istedenfor å hjelpe studenter. En løsning på dette problemet er å automatisk generere variantoppgaver til bruk både i øvinger og eksamener.

Målet med denne avhandlingen var å designe og lage en prototype for generering av variantoppgaver beregnet på digitale programmeringseksamener og øvingsoppgaver. Masteroppgaven evaluerer også hvordan prototypen oppleves, både med hensyn til oppfattet brukervennlighet hos professorer og andre i fagstab, og et utvalg studenters oppfattelse av de genererte variantoppgavene. Systemet benytter seg av store språkmodeller utviklet av OpenAI som tilhører deres GPT-rekke av modeller. Systemet har som mål å øke effektiviteten i emner ved å automatisere oppgaveskapning i stor skala, samt å tilgjengeliggjøre kreativiteten som er å finne i store språkmodeller for fagstaben.

Avhandlingen har sitt utspring i en prosjektoppgave, der det en prototype for å generere varianter av programmeringsoppgaven be utviklet. Etter å ha utviklet den nevnte prototypen og undersøkt svakhetene ved et slikt system, endret prosjektet retning til å benytte seg av nye og moderne språkmodeller i håp om å gi bedre resultater. Gjennom iterativ utvikling og sluttbrukertesting ble systemet vurdert til å ha noen fordeler i forhold til å lage programmeringsoppgaver manuelt, og til å være et effektivt hjelpemiddel for å lage programmeringsoppgaver som en kan bygge videre på. Systemet har flere mangler, men som prototype er det lovende og skisserer en fremtid der store språkmodeller kan brukes til å pålitelig generere kreative variantoppgaver som oppfordrer til læring.

Preface

This thesis marks the end of our five years at the Norwegian University of Science and Technology. We would like to extend our gratitude towards our thesis supervisor Guttorm Sindre, and both students and professors that participated in the testing of our prototype. We would also like to thank our family and fellow students for supporting us through the years, and B.B. for encouragement.

Contents

Figures

Tables

Code Listings

Chapter 1

Introduction

1.1 Motivation

With the ever-growing prominence of programming in our digital age, the demand for quality education in the field of programming has witnessed a significant surge. This increase in popularity inevitably leads to an increased pressure put on faculty staff to create programming coursework. Among the main challenges of teaching programming to students is creating relevant, stimulating and engaging programming tasks for students to complete. Traditionally, programming tasks are crafted manually by an instructor, something that requires deep knowledge and a robust tenure of experience within the field. When creating such tasks, instructors must take into account a myriad of factors such as the students initial skill levels, student skill level as the course progresses, and the need to encapsulate programming principles vital for succeeding in subsequent courses. This task is very complex, and demands a substantial investment of time.

In the traditional academic setting, the substantial effort required to develop high quality programming tasks often leads course faculty to create a single variant of an assessment, distributing the same tasks to all students in a course. The practice of handing all the students copies the same coursework brings with it some challenges. Namely, some students tend to "avoid" understanding programming concepts by utilizing shortcuts such as memorizing solutions without proper understanding, or directly copying work from their peers[1]. While the prevalence of cheating varies greatly depending on a large number of factors, instances of cheating have been reported to be alarmingly high, reaching rates up to 83% in certain contexts[2]. Students resorting to copying the work of others instead of getting valuable practice will suffer serious consequences regarding their learning, something that has been previously emphasized by students as being one of the biggest causes of failure when learning programming languages[3].

Recent advances in the field of Natural Language Processing (NLP) have facilitated the development of software based on Large Language Models (LLMs), such as ChatGPT[4]. These advanced technologies are capable of generating high-quality, contextually relevant natural language text and code, based on natural language input. The demonstrable success of such language models in generating text has led to increased interest in using them for a wide range of applications, including generating programming tasks. Utilizing LLMs for the creation of both coursework and exams presents a compelling prospect, as it has the potential to offer an unprecedented level of variability and creativity in the tasks, while simultaneously exhibiting time-efficiency and scalability.

Based on this, the automatic generation of programming tasks that contain similar concepts, but with individual variations, could offer pedagogical benefits while simultaneously deterring academic dishonesty. By providing all students with unique assignments, the opportunity for word-by-word copying ceases to exist as a whole. This approach would not merely address the symptom of academic dishonesty, but rather targets its root cause by making the act of copying non-viable and encourage actual learning.

Baist et al.[5] conducted research aiming to identify which difficulties are faced by novice students when creating computer programs. Their findings reveal a significant struggle among beginner students in grasping even the foundational concepts of programming structure. Upon inquiring about the perceived effective methods of learning programming, "Working alone on programming coursework" achieved a mean score of $3,06$ with $\sigma = 1,00$ on a scale from $1-5$. This implies a portion of students believe that individual work on programming coursework enhances their learning experience. Furthermore, it intuitively aligns with the notion that consistent practice makes one better at programming. Similar research show the same conclusions, emphasizing the need for students to do programming by themselves[6] in order to improve their skill. Using a system for automatically generating varied programming tasks could present students with an almost inexhaustible array of programming challenges, enabling expansive practice opportunities with minimal demand on the course staff.

The use of LLMs in automated generation of programming variant tasks stems from a pre-project related to this thesis, a literature study related to generating variant tasks and developing a simple prototype capable of generating very basic code snippets. During the testing phase of the prototype, it became clear that the simultaneous creation of meaningful programming tasks, minimal required user input, and significant task variation presented a considerable challenge. More sophisticated technologies and strategies were needed for automated task generation. As we were working on the project, the popularity of LLMs kept increasing and really took off as ChatGPT[4] was released in November 2022. The choice to use a language model for the generation of programming tasks was primarily driven by the objective to enhance the variability and perceived creativity of tasks generated by our task generator, which was developed during the pre-thesis project.

1.2 Research Questions

The main focus of the project is to look at ways of utilizing large language models for generating programming exercises with minimal user input. We further seek to identify the best approach for evaluating the complexity of tasks created, and examine different approaches for AI-driven task creation. The project focuses on CS1 courses in general, but we specifically utilize data from the main CS1 course at NTNU. As a result of this, three research questions that examine our primary objective have been outlined:

Research Question 1 (RQ1): How can a software system be structured in order to allow effortless generation of programming tasks using OpenAI GPT-based language models to be readily applicable?

Research Question 2 (RQ2): What are the main challenges of developing a system meant to generate and assess the complexity of programming tasks?

Research Question 3 (RQ3): What different use-cases does automated generation of variant tasks have, and in which settings does it perform best?

1.3 Report structure

This paper describes our approach to creating a framework for automatically generating programming tasks using large language models. We explore how using a language model affects the variation and perceived creativity of programming tasks, while examining the challenges associated with using language models for this purpose.

Our report is organized into several sections, each dedicated to a different aspect of our research. Introduction sets the stage by detailing the motivation behind the project, providing a summary of the pre-master project and explaining the pivot made to the master project.

Section 2 explains the conceptual foundations of our thesis, presenting the state of our pre-master prototype, and theory relevant for the development of our system. It also contains an in-depth exploration of language models and neural network architectures, with a main focus on OpenAI's GPT-series.

Section 3 details the process of developing the solution, our implementation strategy and evaluation of the system.

Section 4 section sheds light on relevant work that influenced our work both before and during development, with the main focus being on the use of language models to automatically generate programming tasks.

Section 5 presents our results, featuring our variant-task generation and rating system. The section also contains a breakdown of the system from the perspective of a user, followed by an assessment of the system based on different evaluation criteria.

Section 6 contains reflections and interpretations we have made regarding our system, and how our results can be interpreted in light of related work.

Section 7 section contains the conclusion we draw from our thesis. It also discusses the implications of our research, and provides suggestions for future work.

Chapter 2

Background

2.1 State of pre-master project prototype

In the pre-master project[7] related to this thesis, two initial iterations of the artifact prototype were constructed, both without the use of NLP. In short, these versions of the program utilized predefined templates to generate code snippets called variants through a step-by-step generator. The output from the predefined templates served as a basis for programming tasks, from which either task descriptions or task solutions could be created. The purpose of each generated variant was to represent a programming task, either as part of the task description or as a solution to the task.

Each variant was represented as a set of skeletons, through an underlying skeleton framework designed to organize code blocks in a dynamic and scalable way. The skeletons of each variant were nested in a tree structure such that the root was able to build the complete python code. An example of using the skeletons to represent a specific python function is seen in Listing 2.1.

Code listing 2.1: Usage of the pre-master project system: Building a target function using "skeletons"

```
target = f'''def someIterations():
   n = 1
   while n < 5:
       print("Next iteration..")
       n = n + 1
   return n
'''

root = RootSkeleton()

func = Function(parent=root, name='someIterations')
var = VariableDeclaration(parent=func, key='n', value=1)
func.add_to_returns(var)

loop = WhileLoop(parent=func, left=var.get_key(), comparator="<", right=5)
FunctionCall(parent=loop, function='print', arguments=["Next iteration.."])
Assignment(parent=loop, left=var, right=f'{var.get_key()} + 1')
```

The templates used were defined by end users of the system, and stored in a database. They contained a set of elements such as functions, loops and if-conditions, all corresponding to a skeleton in the mentioned skeleton-framework. The generator made assumptions on which order to generate each element, thus resulting in the tree-structure. This approach is mainly deterministic, with output almost one-to-one with the template used. The most significant varieties introduced were randomized variable names and values, and the ordering of loops and if-conditions within functions. A "VariableGenerator" class kept track of the variables used.

The template-format posed challenges for the task author in terms of setup. Each element within the template necessitated the inclusion of corresponding parameters. As shown in Listing 2.1, the loop is instanced with a condition on when to break, and the assignment is provided with the variable and new value. These parameters came from the specifications on each element in the template, and required the author to consider the logic line-by-line, much like writing ordinary code. Moreover, the resulting variants generated through this approach exhibited limited diversity. Consequently, there arose a need for an alternative approach that minimized the author's input while increasing the entropy for each step in the generation process.

The concluding section of the pre-master project outlines three main areas of development for the task generating system. Firstly, the system required more variation and randomization in task generation, the most obvious approach being incorporating NLP. Secondly, the task metrics identified in the study needed to be implemented in the system as evaluators so that authors can be provided with ratings on perceived difficulty and complexity of each task. Finally, the system should support automatically formatting task according to desired formats, specified through a frontend GUI. The generation of textual task descriptions should be automated, adapting the structure and wording to task formats.

2.2 Programming task formats

The assessment of programming task difficulty and complexity is heavily influenced by the format or type of task. Various formats exist, including code-writing tasks, multiple-choice questions, and tracing problems. The format chosen for a task impacts both the cognitive load required for its completion and the resulting learning outcome, although a clear correlation between the two is not always apparent. Tasks that provide students with initial code to work with, as opposed to requiring them to write a program from scratch, may yield similar learning outcomes while being considered "easier." In support of this, Iyer and Zilles[8] refer to Van Merriënboer's suggestion of completing partially written programs as a lower cognitive load activity. Some task formats allow for multiple correct answers, while others have only one definitive solution, and both scenarios can potentially increase or decrease task difficulty.

While the effectiveness of each task format will vary depending on the specific case, it is important for students learning programming to be exposed to various coding scenarios and formats. This principle is reflected in typical CS1 exams, where a combination of task formats are often employed. Restricting students to a single format may lead to the recognition of patterns specific to that format, potentially hindering their development of creative problem-solving skills. During the pre-master project[7], suitable task formats were carefully selected by analyzing their strengths and considering the time required for their creation and assessment. Proceeding with these formats is deemed sensible when pivoting to an NLP-approach.

2.2.1 Selected programming task formats

The advantages of incorporating multiple task formats in assessment highlight the need for a task generation system that can accommodate diverse formats. Supporting different formats is a way of exploring multiple use-cases of automated variant-tasks generation, as described in RQ3[1.2]. To define a scope, six primary task formats were selected:

Code-writing according to a description: This is a common task format when testing overall understanding and multiple concepts at once. Tasks often have a specific method in mind, and could provide example of execution with output.

Example: Write a method **fibonacci()** that calculates the fibonacci number of an input number.

```
print(fibonacci(8))
>>> 21
```

Find the code lines containing errors This format requires the student to understand the code, and critically analyze each line. Distractors are used to replace one or several correct code-lines.

Example: Find the two errors in the method **find-max()**.

```
find_max(list):
    max = 10

    for num in list:
        if num < max:
                max = num
    return max
```

Fill in the blank line of code: This format provides the student with basis, and requires understanding the provided code to solve the missing pieces.

Example: Finish the method **calculate-mean(list)**.

```
calculate_mean(list):
    sum = 0

    for num in list:
        sum += num
    return [..]
```

State the output (program tracing): This format requires the student to follow the logic of the code, however it is not necessary to completely understand all underlying concepts.

Example: What would the function call **myst([12, 4, 13])** output?

```
myst(list):
    result = 2

    for n in range(len(list)):
        result = (result + list[0]) % list[1]
    return result
```

Select and organize code-lines (Parsons problem): This format requires the student understand what the program should do, but not necessarily all details of each line.

Example: Build the method **find-max()** using six of the provided code-lines.

```
#1  find_max(list):
#2  if num < max:
#3  for num in list:
#4  if num > max:
#5  for num in range(list):
#6  max = num
#7  max = 10
#8  max = 0
#9  return max
```

Multiple-choice: The multiple-choice format provides the student with several alternative solutions, of which only one is correct. The goal is to identify the correct solution. The format requires a method of generating plausible distractors.

Example: Which function solves the following problem: Taking a list of numbers as input and returning the sum of all odd numbers.

```
# a)
def sum_odd_numbers(numbers):
    sum_odd = 0
```

```
    i = 0
    while i < len(numbers):
        if numbers[i] % 2 == 0:
            sum_odd += numbers[i]
        i += 1
    return sum_odd
# b)
def sum_odd_numbers(numbers):
    sum_odd = 0
    i = 0
    while i < len(numbers):
        if numbers[i] % 2 == 1:
            sum_odd += numbers[i]
        i += 1
    return sum_odd
# c)
def sum_odd_numbers(numbers):
    sum_odd = 0
    i = 0
    while i < len(numbers):
        if numbers[i] % 2 != 0:
            sum_odd += numbers[i]
        i += 1
    return sum_odd
# d)
def sum_odd_numbers(numbers):
    sum_odd = 0
    i = 0
    while i < len(numbers):
        if numbers[i] % 2 > 0:
            sum_odd += numbers[i]
        i += 1
    return sum_odd
```

2.2.2 Other relevant programming tasks formats

An innumerable amount of other programming tasks exists, but were disregarded in the pre-master project. A few of these formats are still interesting when considering an NLP-driven approach to task generation:

Extend or modify code according to a description: This format is often employed when teaching larger and more comprehensive concepts. The student needs to understand the provided code and the ideas behind it, and expand on them to solve the problem.

Example: Modify the method **sum_numbers()** so that it only sums positive numbers.

```
sum_numbers(list):
    sum = 0

    for num in list:
        sum += num
    return sum
```

Case tasks: This format has a more complex task description than those previously mentioned, and requires concise textual description and context. Because of its complexity and implications, the format requires a lot from a task generation system.

Explain a program with words: This format is well suited to test code comprehension, but not as relevant to automatic assessment in this context.

2.3 The link between code complexity and perceived code difficulty

This section is based on the pre-master project section regarding the link between code complexity and code difficulty. It explored the viable ways of estimating perceived coding task difficulty, with the relevant concepts included below.

People are different, and this diversity extends to students, professors, and other people's perception of what constitutes difficulty. In the context of an introductory university course, the level of expertise among students can range from those who are absolute novices with no prior experience to students which have already started their programming education by watching videos online or learning programming language syntax.

Whatever their starting points are, as students go through an introductory course in programming, they experience an enhancement in their conceptual understanding skillset and knowledge. This enhancement is usually met with increasingly difficult coursework supplied by the course staff[9]. However, as the possibilities of individually customized coursework becomes more available due to automatization, finding accurate measures of task difficulty is important.

The definition of difficulty and the definition of complexity varies depending on the author, but the middle-road is that task complexity is a combination of intrinsic properties while task difficulty is individual, and is dependant on the user interacting with the task. Task difficulty estimation is therefore dependant on gathering data from users solving a task, while a representation of task complexity can be calculated from the task itself.

Pelánek et al.[10] explore the closely related but distinct concepts of complexity and difficulty, with a focus on intelligent learning systems. They used the code made by over 800 students in two Python-based programming courses to examine the most effective heuristic for student development and progress throughout courses. They found that traditional task metrics does not correlate well with time passed in a course, and instead measure that metrics based on syntactic complexity and solution size correlate strongly with time spent in a course.

As courses progress, students are able to solve problems they previously thought too difficult. As current programming courses are taught, exercises are design to explain and teach new concepts. As the course progresses, new concepts are introduced to exercises, while the old concepts are still used in the exercises. Therefore, as the course progresses, exercises contain more and more concepts, which also requires more and more lines of code, which both appears to correlate with task difficulty.

Pelánek et al.[10] define item *complexity* as "... an intrinsic item characteristic, which aggregates item aspects that influence how students solve the item. Complexity is concerned with the structure of the item itself." - Pelánek et al.[10](p. 202). They define item *difficulty* as "... how hard it is to solve the item for students." - Pelánek et al.[10](p. 202). These definitions are well established and much used within the professional environment.

One of the primary challenges in identifying task difficulty lies in the inherently subjective nature of such measurements. The degree of difficulty attributed to a task is dependant to the individual being assessed, thus limiting its applicability across different people performing the same task. Therefore, a single difficulty measurement derived from the interaction of one person with a task lacks universal relevance. For such measurements to be useful in the context of measuring the "average" (or other key metric) perceived difficulty level of a task, the correlation between multiple people's course performance, their success in solving a task, and the properties of the solved task must be measured. This approach would provide a substantial data foundation, which when properly analyzed could offer meaningful insights regarding task difficulty. It is however considerably more complex to gather the dataset needed than measuring task complexity, and using this as a basis instead.

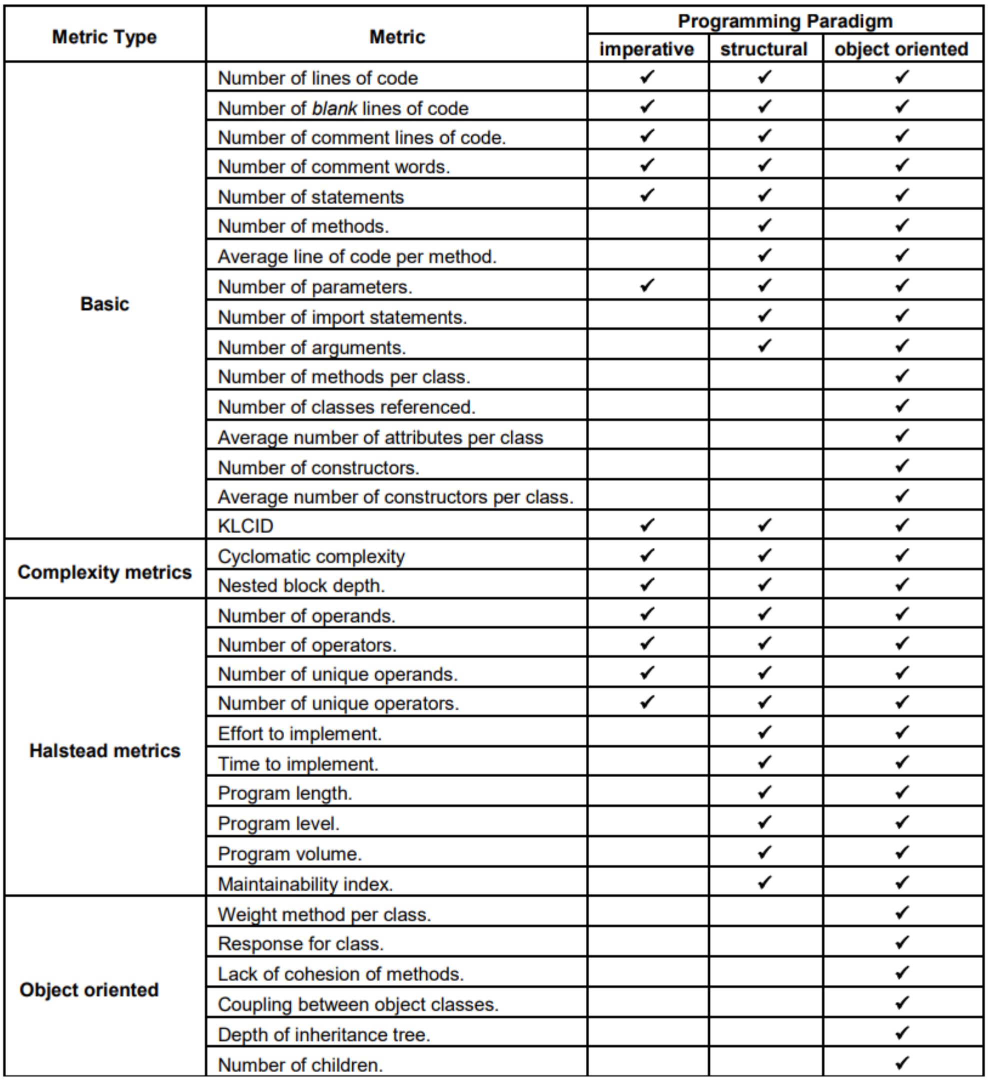
Liu et al.[11]'s task complexity framework "complexity dimensions", where three of them are size, variety and relationship. Exactly these three complexity dimensions are pointed to by Pelánek et al.[10] as being directly relevant to educational items. When analyzing the complexity of a code snippet, especially the complexity dimensions of size and variety are relevant as it can be argued that they are distinctly more measurable than the other dimensions.

2.3.1 Estimating task complexity

Estimating complexity in programming tasks can be achieved without user interaction by considering various identifiers such as the number of code lines required, the cyclomatic complexity of the resulting code, and the inclusion of multiple concepts. Kasto et al.[12] measured the difficulty of code comprehension tasks by using some the software metrics they deemed most appropriate from figure 2.1.

The tasks investigated were specifically code tracing tasks, and "Explain in Plain English"-tasks (EiPE tasks), where they chose to utilize the following metrics:

- Number of statements
- Number of operands (including all identifiers that are not key words)
- Cyclomatic complexity
- Average nested block depth
- Average number of parameters



Metric Type	Metric	Programming Paradigm		
		imperative	structural	object oriented
Basic	Number of lines of code	✓	✓	✓
	Number of *blank* lines of code	✓	✓	✓
	Number of comment lines of code.	✓	✓	✓
	Number of comment words.	✓	✓	✓
	Number of statements	✓	✓	✓
	Number of methods.		✓	✓
	Average line of code per method.		✓	✓
	Number of parameters.	✓	✓	✓
	Number of import statements.		✓	✓
	Number of arguments.		✓	✓
	Number of methods per class.			✓
	Number of classes referenced.			✓
	Average number of attributes per class			✓
	Number of constructors.			✓
	Average number of constructors per class.			✓
	KLCID	✓	✓	✓
Complexity metrics	Cyclomatic complexity	✓	✓	✓
	Nested block depth.	✓	✓	✓
Halstead metrics	Number of operands.	✓	✓	✓
	Number of operators.	✓	✓	✓
	Number of unique operands.	✓	✓	✓
	Number of unique operators.	✓	✓	✓
	Effort to implement.		✓	✓
	Time to implement.		✓	✓
	Program length.		✓	✓
	Program level.		✓	✓
	Program volume.		✓	✓
	Maintainability index.		✓	✓
Object oriented	Weight method per class.			✓
	Response for class.			✓
	Lack of cohesion of methods.			✓
	Coupling between object classes.			✓
	Depth of inheritance tree.			✓
	Number of children.			✓

Figure 2.1: Figure taken from Kasto et al.[12](p. 60). The table shows a diverse selection of static metrics, and their applicability across programming paradigms. The correlations are between tracing tasks, and the metrics.

Note that the metric "Number of statements" was changed to the metric "Number of lines". This was due to the verbose nature of the Java language, which was the language used in the paper. Many lines simply consisted of opening and closing brackets, which were deemed not contributory to the complexity of the tasks. The metrics were calculated using Rationale Software Analyzer[13], and compared to student performance on the given tasks using 2-tailed Kendall's T-b with $p < 0,05$. None of the metrics they used correlated significantly with EiPE tasks, but table 2.1 shows the metrics from code tracing questions that were significantly correlated to the student performance, and therefore to the observed difficulty of the question.

Software metric	Kendall's T-b (2-tailed)
Cyclomatic complexity	0,775
Average nested block depth	0,775
Sum of all operands in the executed statements	0,732
Number of commands in the executed statements	0,732

Table 2.1: Figure derived from Kasto et al.[12](p. 63). The table shows only correlations with $p < 0,05$ between software metrics and question difficulty. The results are interesting in the context of exploring how to measure the complexity of a programming task.

However, they do point out the fact that the paper does not take into account the context of tasks, which significantly impacts perceived difficulty.

2.4 Neural network architecture, language models, and the history of GPT

This section discusses the history of language models and neural network architectures, OpenAIs GPT models, their capabilities at the time and how it evolved to its current state.

Generative Pre-trained Transformer (GPT) is a series of autoregressive language models made by OpenAI that uses deep learning to produce human-like text[14]. It has many variants, from GPT-2 and GPT-3 which are models that can understand and generate natural language, to Dall-E, which can generate and transform images based on natural language input[15]. GPT-3.5 and GPT-4 (still in beta as of June 2023) are interesting in the context of task generation, as they're able to understand and generate both natural language and code.

2.4.1 Popular language model architectures

The transformer model

This section briefly explains the workings of the transformer model, which is the base of some of the groundbreaking modern large language models.

The Transformer model is a neural network architecture introduced by Vaswani et al. [16] in 2017. The key innovation of the architecture is the use of self-attention mechanisms, allowing the model to focus on different parts of the input sequence at each step of the computation, enabling the capture of long-range dependencies and vague context. It also allows the model to account for the importance of different words in a sequence. The architecture uses many of these self-attention layers, that each focus on a different subspace of the input sequence. This allows the model to capture different aspects of relationships between words. Figure 2.2, taken from Vaswani et al. [16] shows the detailed architecture of the Transformer architecture.

The transformer model consists of an equal number of encoders and decoders (6 of each in the original paper). Both the encoder and decoder are composed of stacked layers of self-attention and feed-forward neural networks. The encoder takes an input sequence, and processes it to produce a sequence of hidden representations. The decoder interprets these representations and generates the output sequence. During self-attention, the model computes the attention scores for each pair of words in the input sequence. These scores determine the importance or relevance of one word to another. Jay Alammar[17] illustrates this well with a figure, seen in figure 2.3. He uses the translation of the sentence "The animal didn't cross the street because it was too tired" from English to French as the task. The word "it" is used in the translation, but what is "it" referring to? Self-attention allows the model to look at other words (positions) in the input sequence to better represent what "it" refers to. Figure 2.3 illustrates this, and shows that "it" is most strongly associated with "animal" in this sentence. Understanding what "it" refers to in this sentence is trivial for human beings, but is anything but that to a computer. Further readings on the transformer model can be found here[17][18].

The transformer architecture revolutionized the Natural Language Processing field and has since been widely adopted in various NLP applications. It currently serves as the basis for several state-of-the-art large language models, as explained below.

Autoregressive language models

The following section aim to provide a surface level explanation of the workings of an autoregressive model, and provide examples of successful language models based on autoregressive technology.

As previously mentioned, GPT is an autoregressive language model. This kind of model is a probabilistic model that generates sequences of words by estimating the conditional probability of each word appearing next, given the previous words in the sequence. This is in line with the concept of conditional independence, which assumes that the probability of a word in a sentence only depends

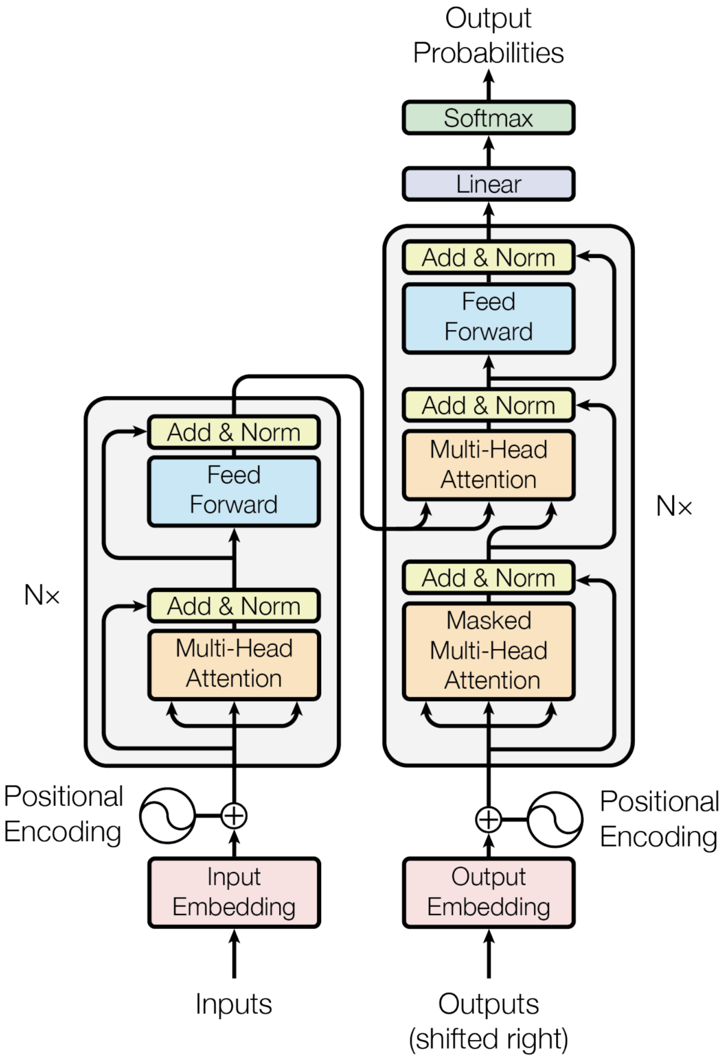


Figure 2.2: Taken from Vaswani et al.[16]. The leftmost part of the figure is an encoder, and the rightmost part is a decoder. Embeddings are the first layer of the Transformer model, which convert words from the input sentence into vectors that represent each word in a high-dimensional space. For the model to be able to consider the position of words in the sentence, positional encodings are added to the embeddings. Multi-head attention is a module for attention mechanisms running in parallel. This allows each word to understand the context in which it appears, and is given a score indicating its "importance". The box labeled "Feed forward" represents feed-forward neural networks, where each input position is connected to each output position by the neural networks. The networks consist of two linear transformations with a ReLU activation inbetween. Add & norm represent normalization and residual connections, used to help train the model effectively and to mitigate the common problem of vanishing gradients. These components are stacked together multiple times (6 in the original paper) to form the Encoder part of the transformer. Models like OpenAIs GPT family use a variant of this architecture, as does Googles BERT.

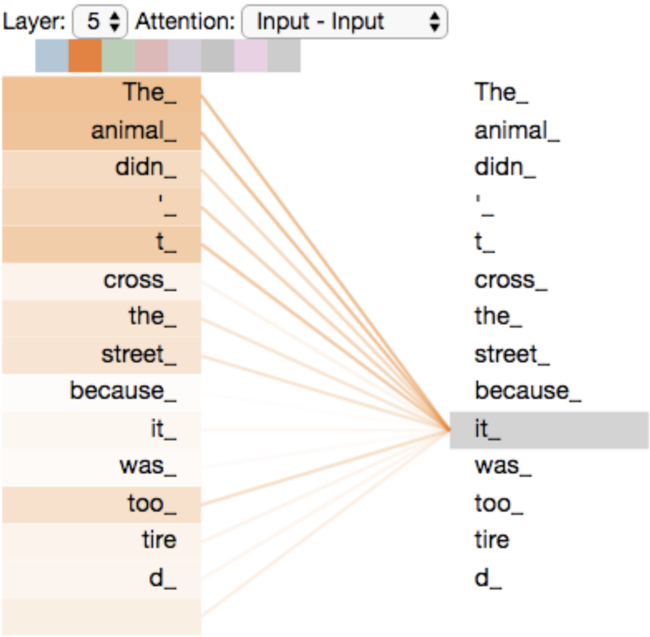


Figure 2.3: Taken from Alammar[17]. The figure shows the concept of self-attention, and models how strongly each word in the sentence is associated with every other word in the sentence at layer 5/5. Full Collab sheet found here[19].

on the preceding words in the word sequence. In other words, an autoregressive model will process an input, and extrapolate what it believes is the most probable response to a natural language query.

For an autoregressive language model to process queries this way, the models are typically trained using "maximum likelihood estimation"[20]. Large datasets consisting of sequences of words (sentences) are handed to the model. The training involves predicting the next word in a sequence given the previous words, where the likelihood of a sentence appearing is calculated as the product of the probabilities assigned to each word in the sequence. The training objective is to maximize the average log-likelihood of the training sequences, which again is equivalent to minimizing the negative log-likelihood. The reasoning behind using log-likelihoods instead of the actual likelihoods, is simply due to the scale of numbers. Multiplying many small probabilities together usually result in very small numbers. If one instead takes the logarithm of the likelihoods and add them together, it both simplifies the computations and leads to higher numerical stability.

By maximizing the log-likelihood, the autoregressive language model is encouraged to assign higher probabilities to words that are likely to occur in the training data, hopefully improving its ability to generate relevant and coherent text.

Other autoregressive language models

OpenAI are not the only ones capable of creating successful autoregressive large language models. Touvron et al.[21] from Meta published a paper the 27th of February 2023 regarding LLaMA (Large Language Model Meta AI), a collection of base language models ranging from 7B to 65B parameters. The models are trained on trillions of tokens, and differ from other language models due to the fact that they are trained on publicly available datasets exclusively, not making use of proprietary or otherwise inaccessible datasets. LLaMA is based on a modified transformer architecture, with the 13B parameter version outperforming the 175B parameter version of GPT-3 on most benchmarks, and is even competitive with some of the current best models such as DeepMinds 70B parameter Chinchilla[22] and Google AI's 540B parameter PaLM[23] LLM, of which both are autoregressive language models. If the current trends are to continue, the mentioned language models will be considered small in a few years.

Bidirectional language models

As the name suggests, bidirectional language models do not only look at the preceding words in a sequence when finding the max probability word. Instead, it tries to capture dependencies in both directions of a sequence by considering the context of a word from both the preceding and succeeding words. This is done by processing the input sequence both forwards and backwards. This way, the model considers the context from both directions, which in turn allows it to capture a broader range of dependencies. It naturally follows that bidirectional language models are considered more useful for tasks where the meaning of words is more diffuse, and depend more on the words surrounding context.

BERT (Bidirectional Encoder Representaions from Transformers) is a bidirectional language model that was developed by researchers at Google AI Language, first introduced by Devlin et al.[24] in the paper "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding" first published in 2018. BERT was trained using a two-step process of pre-training and fine-tuning. Its training is similar to GPTs training, as the primary goal of pre-training is to teach the model to predict missing words in sentences. Just as GPT, BERT is self-attentive which allows it to capture relationships between different words in the input sequence.

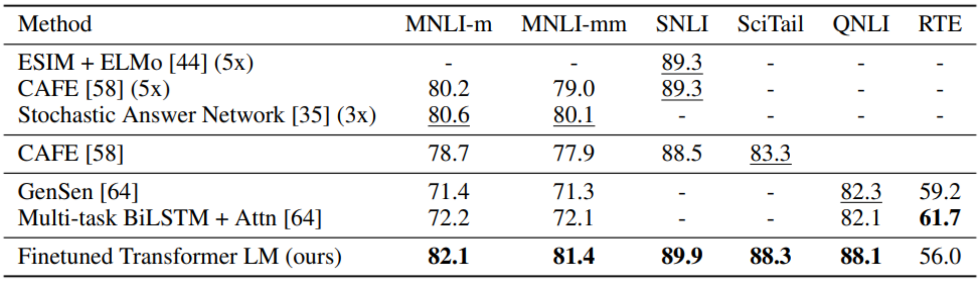
BERT++ was introduced by Tenney et al.[25] in 2019, and is an extension of the original BERT model. BERT++ aimed to improve upon the limitations of BERT by incorporating linguistic information and fine-grained modeling of the classic NLP pipeline. The improved model also contains additional components such as dependency parsing embeddings, and semantic role labeling embeddings. These improvements help the model capture syntactic and semantic information more effectively.

Current language model use-cases

ChatGPT[4][26] is a web application exposing different versions of OpenAI's language models to end users. The language models are variants of the GPT series which is a class of deep learning model developed by OpenAI. The autoregressive language models used in ChatGPT are specifically focused on generating human-like responses in a conversational context. After ChatGPT gained massive popularity online, Google announced they were working on a chatbot similar to ChatGPT driven by a powerful language model, called Bard. However, it has not yet been released to the general european public.

2.4.2 GPT-1

Before transformer models, text generation was usually performed with other deep learning models such as RNNs (Recurrent Neural Networks) and LSTMs (Long Short-Term Memory networks). Radford et al. from OpenAI published the paper Improving Language Understanding by Generative Pretraining[27] in 2018, which describes the development of a Generative pre-training method for language understanding tasks using transformer-based neural network architecture. As the pre-training unsupervised, this method is dramatically cheaper than having to purchase or make labeled datasets. The pre-trained model in question was fine-tuned to handle question answering and text classification, with comparisons towards the current state-of-the-art methods showing drastic improvements as seen in Figure 2.4.



Method	MNLI-m	MNLI-mm	SNLI	SciTail	QNLI	RTE
ESIM + ELMo [44] (5x)	-	-	89.3	-	-	-
CAFE [58] (5x)	80.2	79.0	89.3	-	-	-
Stochastic Answer Network [35] (3x)	80.6	80.1	-	-	-	-
CAFE [58]	78.7	77.9	88.5	83.3		
GenSen [64]	71.4	71.3	-	-	82.3	59.2
Multi-task BiLSTM + Attn [64]	72.2	72.1	-	-	82.1	**61.7**
Finetuned Transformer LM (ours)	**82.1**	**81.4**	**89.9**	**88.3**	**88.1**	56.0

Figure 2.4: Quoted from Radford et al.[27]: "Experimental results on natural language inference tasks, comparing our model with current state-of-the-art methods. 5x indicates an ensemble of 5 models. All datasets use accuracy as the evaluation metric." Full figure and more tables are found on Radford et al.[27](p6.)

GPT-1's zero shot performance is also important to note. A zero-shot setting refers to the ability of a language model to perform a task it has not been explicitly trained on. If a model has a high zero-shot task performance, it could solve a variety of tasks it has not been explicitly trained to solve. Such a model would be extremely useful in all aspects of modern life, from education, politics to medicine. Figure 2.5 below shows GPT-1's evolution of relative task performance as a function of LM pre-training updates.

2.4.3 GPT-2

A year later in 2019, Radford et al. from OpenAI published the paper "Language Models are Unsupervised Multitask Learners[28], wherein they discuss GPT-2, their (then) largest language model now boasting 1.5 billion parameters. The authors argue that the large dataset used in its unsupervised pre-training enables it to outperform supervised pre-training models simply due to its exposure to rich natural language. GPT-2 was made available to the machine learning community, and found some use for text generation tasks.
Especially zero-shot settings showed great promise, with GPT-2 zero-shotting to state of the art performance on 7 out of the 8 tested language modeling datasets.

2.4.4 GPT-3

Brown et al.[29] from OpenAI published yet another paper in 2020 introducing the next generation model, GPT-3. Their now largest autoregressive language model has 175 billion parameters (more than 100x compared to GPT-2), and can perform a variety of tasks such as translation, classification and now reliable lan-

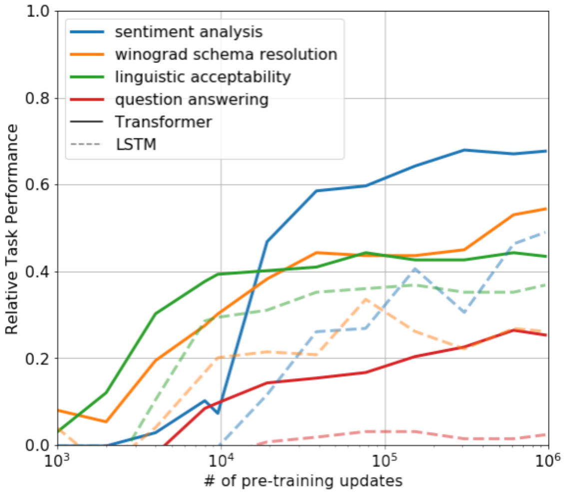


Figure 2.5: "Quoted from Radford(2018) et al.[27] p.7: " Plot showing the evolution of [GPT-1's] zero-shot performance on different tasks as a function of LM pre-training updates. Performance per task is normalized between a random guess baseline and the current state-of-the-art with a single model." Full figure and context is found on p6.

guage generation.

The paper details how scaling up the language model greatly improves task-agnostic few-shot performance, something humans find natural, but NLP systems historically has struggled to succeed in. Even though the model used for benchmarking in the paper has undertaken no gradient updates or fine tuning, the model reached sporadic competitiveness with state-of-the-art fine-tuning approaches at the time, as can be seen in figure 2.6, taken from Brown et al.[29], p.20. The paper also demonstrates the effectiveness of GPT-3 on several benchmark tasks. Figure 2.6 shows model performance on SuperGLUE[30]. SuperGLUE stands for Super General Language Understanding Evaluation, and is a benchmark for evaluating the performance of natural language understanding models. Its language understanding tasks include question answering, natural language inference, and coreference resolution. Note the GPT-3 performance compared to the previously discussed BERT++ model.

As for zero-shot capabilities, GPT-3 significantly outperforms its earlier iterations. However, it is clear that few-shot and even one-shot scenarios significantly boost performance compared to zero shot scenarios, emphasizing the need for adaptation to the task, or that at the very least recognition of the task is vital in order to achieve high performance. Figure 2.7 and 2.8 shows how GPT-3 perform at two different tasks in varying scenarios.

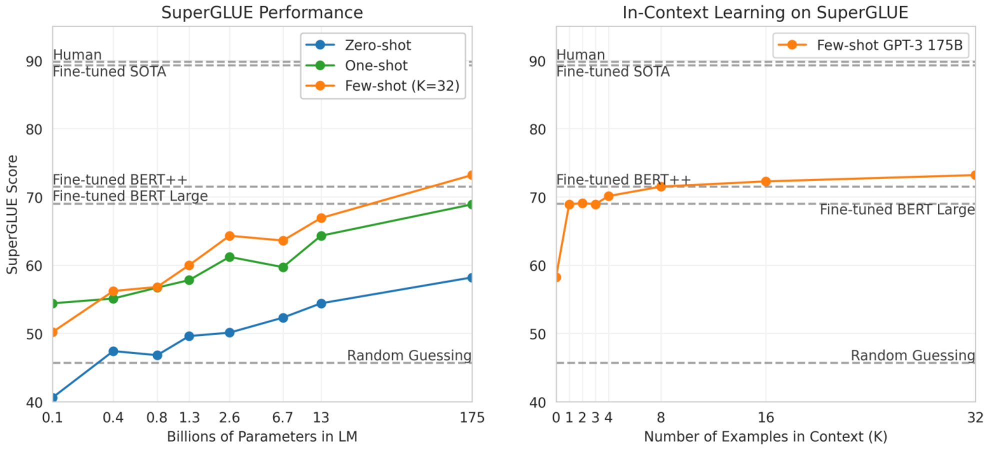
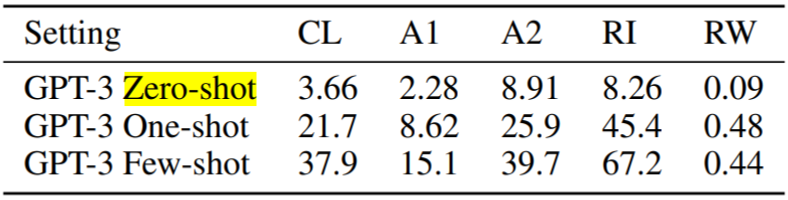


Figure 2.6: Quoted from Brown et al.[29] p. 20: "A value of K=32 means that our model was shown 32 examples per task, for 256 examples total divided across the 8 tasks in SuperGLUE. We report GPT-3 values on the dev set, so our numbers are not directly copmarable to the dotted reference lines. The BERT-Large reference model was fine-tuned on the SuperGLUE training set (125K examples), whereas BERT++ was first fine-tuned on MultiNLI (392K examples) and SWAG (113K examples) before further fine-tuning on the SuperGLUE training set (for a total of 630K fine-tuning examples). We find the difference in performance between the BERT-Large and BERT++ to be roughly equivalent to the difference between GPT-3 with one example per context versus eight examples per context...".

Setting	2D+	2D-	3D+	3D-	4D+	4D-	5D+	5D-	2Dx	1DC
GPT-3 Zero-shot	76.9	58.0	34.2	48.3	4.0	7.5	0.7	0.8	19.8	9.8
GPT-3 One-shot	99.6	86.4	65.5	78.7	14.0	14.0	3.5	3.8	27.4	14.3
GPT-3 Few-shot	100.0	98.9	80.4	94.2	25.5	26.8	9.3	9.9	29.2	21.3

Figure 2.7: Taken from Table 3.9 in Brown et al.[29] p. 23. nD+/nD- is n Digit subtraction or addition. 2Dx is two digit multiplication, and 1DC is 1 digit composite operations. Performance increases significantly from zero-shot to one-shot to two-shot scenarios, but zero-shot performance is generally very good.



Setting	CL	A1	A2	RI	RW
GPT-3 Zero-shot	3.66	2.28	8.91	8.26	0.09
GPT-3 One-shot	21.7	8.62	25.9	45.4	0.48
GPT-3 Few-shot	37.9	15.1	39.7	67.2	0.44

Figure 2.8: Table 3.10 in Brown et al.[29], p. 23. The figure shows GPT-3's 175 billion parameter performance on word unscrambling and word manipulation tasks in zero-, one- and few-shot settings. CL = "Cycle letters in word", A1 = "Anagrams of words except the first and last letters", A2 = "anagrams of words except the first and last two letters", RI = "Random instertion in word", RW = "reversed words". The study emphasizes that adaptation (or at least recognition) to the task is important to performing the computations correctly, as can be seen in the discrepancy between zero-shot and one-shot performance.

2.4.5 GPT-3.5 and onwards

This section offers an overview of the cutting-edge models currently released by OpenAI. Even though the models are quite similar, some are specialized and tailored towards specific use cases. The models and technologies discussed are GPT-3.5, InstructGPT, ChatGPT, Codex and GPT-4.

GPT-3.5

As mentioned above, GPT-3's measured performance was based on a non-tuned version of the model. New versions of GPT-3 were released throughout 2022, both code and natural language focused models, and are described as more capable than the previous versions. Over time, OpenAI began referring to these models as the GPT-3.5 series. Around the same time, ChatGPT[26] was released. This is the software which made natural language AI models accessible to a much wider audience. It was initially based on a GPT-3 model, fine-tuned on a large dataset of conversational data to generate human-like responses to human-like text inputs. In short, GPT-3.5 models are models based on GPT-3, trained on a blend of text and code in 2021 and before. These models include code-davinci-002, text-davinci-002, text-davinci-003, and gpt-3.5-turbo-0301[31].

InstructGPT

Many problems arose as the first iterations of GPT were tested. To understand the problems the researchers faced, it is important to understand the fact that language models such as GPT-3 simply calculate which word is most likely to follow based on the context (previous words). Which words are deemed the most likely are direct results from the training set. A language model may therefore adopt the political viewpoints of training data, or be taught that the pronoun for doctors is "he", and for nurses is "she". These harmful stereotypes will then be echoed by the model. In short, biases in the training data will be reflected in the models' responses. Similar harmful biases has been shown to exist in earlier AI technologies[32][33].

Another problem often encountered when dealing with language models, is mistakes in generated text that is semantically and/or syntactically plausible but is incorrect, disputed or nonsensical. These mistakes in language models are called "hallucinations", and can cause great harm when end users of technologies such as ChatGPT are not aware of its workings. Imagine someone urgently querying ChatGPT for medical advice due to acute health problems, only to be provided with seemingly legit but ultimately bad advice.

As seen in figure 2.9, InstructGPT performs significantly better than GPT-3 when evaluating truthfulness in statements. The same figure also shows how InstructGPT has significantly fewer hallucinations than GPT-3.

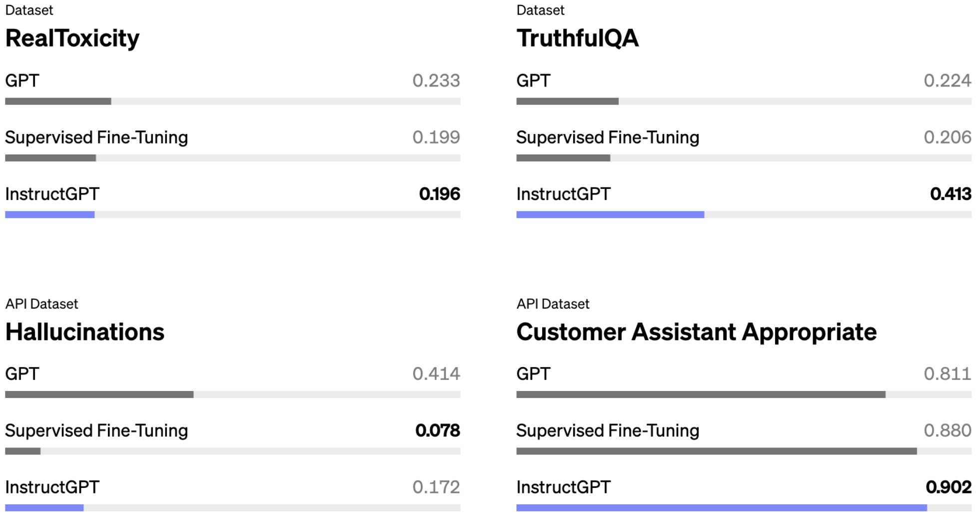


Figure 2.9: Figure taken from OpenAI InstructGPT[34] webpage, and consists of information taken from the following cited sources. Bigger is better for TruthfulQA and Customer Assistant Appropriate, while smaller is better for Hallucinations and RealToxicity. GPT refers to GPT-3. Testing on the TruthfulQA dataset done by Lin et al.[35] shows that InstructGPT performs considerably better when testing truthfulness and level of informativity. According to Gehman et al.[36], InstructGPT is significantly less toxic than GPT-3.

Understanding a user's intent is difficult, especially when intent must be inferred through natural language. Ouyang et al.[37] from OpenAI published a paper in 2022 regarding fine-tuning GPT-3 using supervised learning and fine-tuning the model with human feedback. The paper focuses on language models specifically trained to be better at following user intentions than GPT-3. What follows are more truthful and less toxic models with the ability to follow instructions in a prompt more precisely than previous versions. The model, named InstructGPT, showed significant improvements in toxic output generation. Human evaluations show that outputs from 175B parameter InstructGPT were preferred over outputs from 175B parameter GPT-3 85 ± 3% of the time, and preferred 71 ± 4% of the time to few-shot GPT-3[37]. The model did make simple mistakes(mostly due to the relatively low number of parameters), but showed that including humans in the loop is a good idea to avoid behavioral problems from language models.

As of May 2023 the InstructGPT models, trained with humans in the loop, are deployed as the default language models on OpenAI's API's[34]. Knowing much of their audience lack experience with large language models and knowledge regarding their inner workings, this is a way to reduce possible harm caused by ChatGPT.

ChatGPT

ChatGPT[4][26] is a web application made by OpenAI which provides end users with the ability to prompt some of OpenAIs models using natural language. The term "ChatGPT" can be seen used as a collective term for OpenAIs models in daily speech, or may even be used to refer to modern language models in general. The free version allows interaction with GPT-3.5 based models, while a paid subscription allows access to GPT-4. The current language model used by ChatGPT is fine-tuned from a model in the GPT-3.5 series, and finished training early 2022. The specific language models ChatGPT users interact with is changing constantly, but the common denominator between them is that they're based on the best OpenAI language models, further fine-tuned for conversational interactions. Figure 2.10 shows the method used by OpenAI to train the model, which overall was very similar to how they trained InstructGPT. There were only small differences in the data collection setup. ChatGPT has a wide array of use for end-users, which include everything from code and text generation based on a query, writing assistance, translations and much more. According to Reuters[38], ChatGPT as a service reached 100 million monthly users just two months after launch, and is as of May 2023 the fastest growing consumer application in history.

In June 2020, OpenAI started providing direct access to many of its language models through their API[39]. This allowed developers to make use of OpenAIs language models in their own applications. All language models currently available through their API are built on the same transformer architecture as its predecessors, and utilizes the transformers ability to infer meaning and context from a sentence and provide meaningful replies to natural language text prompts.

Codex

OpenAI's Codex[40] is a large language model specifically tailored towards interpreting and generating code. It's based on GPT-3, but its training data consists of both natural language and billions of lines of source code taken from both GitHub[41] and other sources. This makes the model proficient in over a dozen

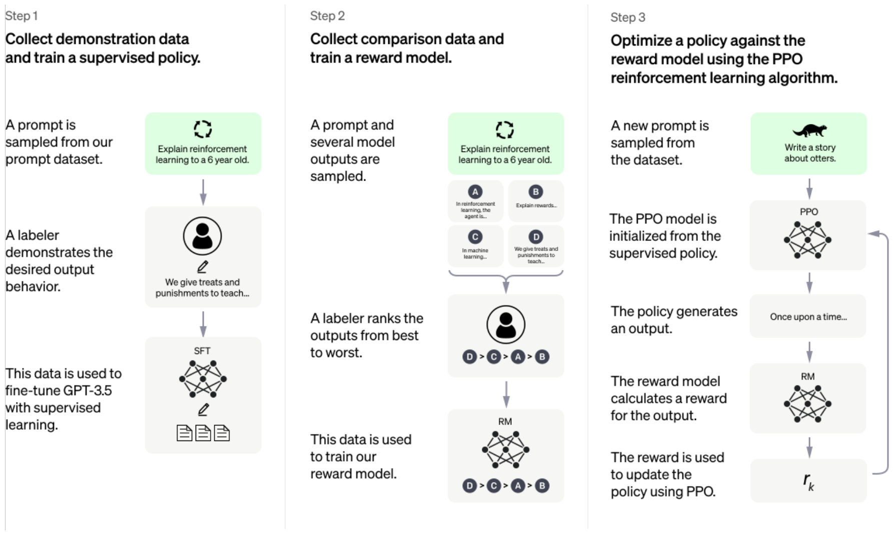


Figure 2.10: Figure taken from ChatGPT website[4], and outlines the method OpenAI used to train the model used in ChatGPT.

languages, but is stated to be the most capable in Python. It is also the model that powers GitHub Copilot[42], an IDE and code editor plugin marketed as an "AI pair programmer" helping users write code faster by utilizing the context of your code to make smart suggestions and recommendations.

2.4.6 GPT-4

GPT-4[43][44], a large multimodal model initially released in March 2023 for closed beta testing, is the most advanced system from OpenAI yet. Performing in the 90th percentile when completing a simulated Bar Examination(the final test a US lawyer must pass in order to be admitted into the bar of a specific jurisdiction), or in the 99th percentile when completing the biology olympiad using computer vision, it significantly outperforms all earlier models. It is capable of accepting both text and image inputs, returning text back to the end user. Due to its recent release, GPT-4's capabilities were not know nor taken into account during most of the development and writing of this project. A wide array of benchmarks for GPT-4 can be found here[45].

Limitations and shortcomings

Even state-of-the-art language models like the ones discussed above suffer from limitations, of which some of them are outlined here.

LLMs such as the GPT models are prone to hallucinations, where the severity and potential harm can vary significantly. For instance, if one queried ChatGPT regarding the total area of a non-existent country, the model might still produce a response despite the country's non-existence. While such instances are misleading, their implications are relatively harmless given the clear fictional premise of the query.

More concerning however, are circumstances where users, expecting factual responses, unknowingly receive hallucinated information. In these scenarios, the user has no immediate way to discover that the model has not provided a factual answer, but instead has filled in gaps based on its training data and pattern completion tendency. Examples such as this emphasize the challenges of using LLMs in contexts where the accuracy of information is crucial.

As LLMs simply function as very large pattern recognizers, their responses can reflect potential harmful ideas or viewpoints that exist within its training data. The models are oblivious towards the content they generate, and have no concepts of right and wrong or understanding of ethical considerations. Their sole aim is to produce responses that are statistically similar to patterns they've learned during training. Given that these models, particularly ChatGPT, are frequently employed as learning aids by audiences including high school students and even younger individuals, the potential impact of such outputs is considerable. Even though OpenAI has made significant progress in largely eliminating harmful information, sheer complexity and volume of potential outputs imply that some undesired responses may still slip through.

It's important to note that LLMs do not "understand" in the same way humans do. Their comprehension of language is purely statistical, based on patterns they have seen in their training data, devoid of the kind of contextual understanding that humans possess. They also lack the ability to understand context in a meaningful manner. They don't possess an awareness of the world or its events, and they can't access or comprehend information beyond what is explicitly provided in the input or the knowledge embedded in their training data. The concept of 'memory' for these models also diverges significantly from human memory. LLMs do not maintain a sustained memory of past interactions or incorporate this into future responses unless explicitly provided. Even then, they tend to struggle when the provided context is too large. For all these reasons a good rule of thumb is to at least be familiar, and preferably have some understanding of LLMs before making use of them.

Chapter 3

Research method

This chapter provides an overview of the methodological considerations underlying our research. Our project presents unique challenges that require a thoughtful process where multiple approaches are relevant.

3.1 Development methodology

3.1.1 Design science

The research paradigm of design science, as described by Hevner et al.[46], provides a systematic approach to the development and evaluation of artifacts within information systems research, such as this study. This methodology, known as Design Science Research (DSR), is designed to create knowledge by constructing and evaluating purposeful designs for solving practical problems.

DSR aims to contribute to the body of knowledge in a domain by producing practical and innovative artifacts that address specific problems. It fosters a close collaboration between researchers and practitioners, bridging the gap between theory and practice in the field of information systems. The methodology is therefore especially relevant when applying knowledge about large language models to automate programming coursework creation.

The DSR framework consists of six main activities that guide the design science research process. To guide the study in this project and acquire an overview of the research process, we consider each activity in relation to key sections of this thesis.

Problem Identification and Motivation: The first step involves identifying and understanding the practical problem that needs to be addressed. This includes a clear definition of the problem and its significance in the context of the domain. The motivation section[1.1] states the benefits of automatically generating programming tasks. RQ1[1.2] narrows the problem down to a framework system that allows effortlessly creating programming exercises using large language models. The results from the pre-master project, as presented in Section 2.1, highlight the incentive to develop an artifact with these properties, in order to decrease time-cost of producing programming exercises while increasing variations.

Define Objectives for a Solution: With the specific problem of task generation identified, it is necessary to define the objectives for the solution. This includes specifying what the artifact should achieve, the desired properties it should possess, and the constraints that need to be considered. Section [3.2] elaborates on this point by discussing different approaches to implementing language models in such a system.

Design and Development: At the design and development stage, the artifact itself is implemented. The design process, detailing the artifact, its components, and its functionalities is scrutinized in Chapter 5. It is beneficial in this stage to draw upon existing theories, models, and best practices to inform the design. In the development of this artifact, both established models such as object-oriented programming and the "Builder"-pattern[47] are adopted, in addition to recently launched state-of-the-art technologies such as OpenAI's GPT-models. To organize the development process, the agile framework described in Section 3.1.2 is used.

Demonstration: After developing the artifact, it needs to be demonstrated and tested on both students and teaching staff, to gain insight into its effectiveness. The demonstration should present the prototype after most of the features are implemented, simulating the intended real world use-cases. The specific demonstrations devised for the project artifact is described in Section 3.4.

Evaluation: The evaluation stage assesses the artifact's utility, quality, and efficacy in solving the identified problem. The evaluation stage is closely related to the demonstration, assessing the findings from both student and teaching staff demonstrations. The evaluation process provides insights into the artifact's strengths, weaknesses, and potential improvements. It is also relevant to compare the developed artifact to existing systems and means of creating coursework, especially comparing generated programming tasks to existing sets of tasks used in CS1 courses. Evaluation of the demonstration is detailed in Section 5.3.

Communication: Finally, the findings and knowledge generated through the design science research need to be structured and communicated. This is done throughout this thesis, documenting the research process, presenting the artifact, and scrutinizing the results.

3.1.2 Agile development

Agile development represents a software development approach that places significant emphasis on collaboration, adaptability, and the capacity to respond to changes. Its primary objective is to deliver software of superior quality swiftly and efficiently by breaking down the development process into smaller, iterative cycles. Our chosen framework for implementing agile development is Scrum[48].

By adopting the Scrum framework, we were able to ensure the progress of our system in an iterative and effective manner. Our sprint length of 4 weeks combined with the principles of collective ownership helped motivate and push the development of the solution forwards, while simultaneously providing the flexibility we needed when it turned out we had made a wrong decision and needed to backtrack.

Throughout the project, we utilized the Scrum board integrated into GitLab [49] as a tool to persistently prioritize tasks during the developmental phase.

Throughout the project, we made use of the Scrum board built into GitLab[49] as a tool to continuously prioritize tasks during the developmentak phase. which helped us keep track of our progress. This tool also provided a visual representation of our progress and served as a monitor of our development velocity, enabling us to learn the realistic times frame required for task completion. In our two-member team, the distribution of traditional Scrum roles was deemed impractical. Instead, we served as compementasry product owners of each other's work, each assuming the role of developer for the other's tasks.

Each sprint was initiated with a planning meeting in which we delineated the goals and tasks to be achieved by the sprint's conclusion.

Every sprint started with a sprint planning meeting, where we established the goals and tasks to be accomplished within the end of the sprint. Upon the completion of each sprint, a review meeting was convened to demonstrate the completed work, and to identify potential areas for improvement.

While this adaptation of Scrum may not align perfectly with the standard interpretation of the framework, our collective experiences within the Scrum framework and our familiarity with each other's work styles enabled us to indentify what strategies would be effective for our small team. This understanding allowed us to incorporate select elements of Scrum into our tailored Agile development framework, thus creating a custom system that addressed our specific needs without unnecessary overhead.

3.2 Approaches to implementing large language models

The initial step of incorporating large language models (LLMs) into task-generation is identifying and comparing possible approaches. In light of the pre-master project, one inherent approach is adapting the existing task-generation system to utilize LLMs at each step in the variant-task generation process. This approach necessitates the development of a method to construct skeletons from the textual output generated by an LLM. This entails simplifying the templates, allowing

LLMs to provide the details. Textual task descriptions could be provided along with the code snippet of the variants. Another approach is discarding the existing templates and skeletons, instead requesting entire variants based on a textual code description. This requires a system for handling the response from the LLMs and formatting tasks accordingly.

By examining more than one approach, it is possible to experiment with advantages and weaknesses of each method, enabling testing their respective qualities in different use-cases. It was deemed likely that each of the approaches would fare differently for each of the task formats listed in Section 2.2.1. For example, tracing exercises could benefit from obscure and random code built incrementally, whereas case-like exercises require the context and coherency provided by requesting an entire variant from an LLM.

3.2.1 Approach 1: Introduce LLMs in the existing system

Starting out with the artifact from the pre-master project, the generation is adapted to use LLM while still following the same logic of generating skeleton by skeleton to make up a variant. The difference is each step utilizing LLMs, making a request for the attributes used in building the skeletons. For instance defining a loop previously required explicitly specifying the variable to iterate over, the type of loop and number of iterations. With the LLM-approach, these decisions are left up to a LLM, introducing more randomness.

Incorporating the LLMs from OpenAPI, spesifically the GPT-models, requires building an "OpenAIClient" class. This class is added to the existing project as a singleton, and makes requests using OpenAPI's python-package. Prompts are built by querying for the element required, drastically reducing the need for users of the system to specify details in the templates. Each step in the LLM generation process has limited access to the broader context of the variant. However, the "VariableFactory" class built during the pre-master project keeps track of the variables in use. For example, one step may request a list variable and receives the following code: *numbers = [1, 3, 4, 5]*. The new variable "numbers" will then be captured by the VariableFactory, stored with the name, value and type. A subsequent step that requests a loop is thus able to check that a variable of the preferred type exists, and iterate over the same list. This system ensures consistency through the variant code, avoiding instancing variables twice. If a required variable does not exist, however, the VariableFactory will create a new one based on either a specified name, value or type.

Despite the structural consistency, and the mechanisms ensuring LLM-built variants are syntactically correct, no structures consider the context or purpose of the logic in the variant. Technically this approach combines many small code-snippets provided by the LLM. The result is more obscure variants where the code

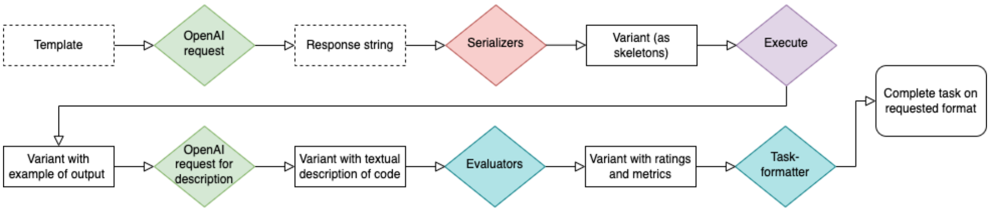


Figure 3.1: Example of steps necessary in generating a variant by implementing LLMs in the pre-project system

solves no specific problem. This generates variants where the focus lies on understanding the specific concepts, such as variable-declarations, loops or conditions, and not on the effects of combining them for problem-solving on a higher level.

3.2.2 Approach 2: Building entire variants using LLMs

The second approach is based on requesting the code snippets for an entire variant, or even several variants, in a single prompt to the LLMs. This method capitalizes on the high quality of output produced by modern LLMs when provided with sufficient context. Instead of requesting smaller elements such as a loop or a variable-declaration, entire functions are requested. This design choice entails building a framework around the mentioned OpenAIClient class, capable of building queries, interpreting responses, and formatting tasks correctly.

Query techniques

To effectively use large language models, formulating high quality queries is important. Queries should provide all necessary context, while staying short and concise. The pre-defined templates were revised to remove most of the list of elements used for specifying skeletons in the pre-master project. It was decided to restrict all templates, and thus all generated variants to functions. Therefore the new template-format only requires a function description. This design choice removes support for classes and variants of multiple functions, but simplifies the queries used when prompting the LLMs. The queries developed for use in the artifact are defined below. Note that the queries are written in English, as OpenAI's GPT-models are shown to perform to better in that particular language, as seen in Figure 3.2. The query for task descriptions, however, is written in Norwegian to ensure responses in Norwegian.

Requesting a single function *Build this python-code with docstring: A function that: [Function-description]*

Requesting a single function from a function-signature *Provide the python-code of a function with docstring from this signature: [Function-signature]*

Requesting multiple functions *Build this python-code with docstrings: [Amount] different functions that all solve the same problem. They [Function-description]*

Requesting multiple functions from a function-signature *Provide the python-code of [Amount] different python-functions with docstrings from this signature: [Function-signature]*

Requesting a task description for a function *Lag en kort oppgavebeskrivelse for en kodeoppgave hvor følgende kode er en mulig løsning, uten å referere til koden spesifikt: [Function code]*

Requesting distractors for a function *Using the following python-code as a base, select one of the lines and create three incorrect alternatives for a programming-exercise: [Function code]. Output the alternative lines, starting with the original line and followed by the three different incorrect alternatives. Format the output as only the code-lines, separated by newlines*

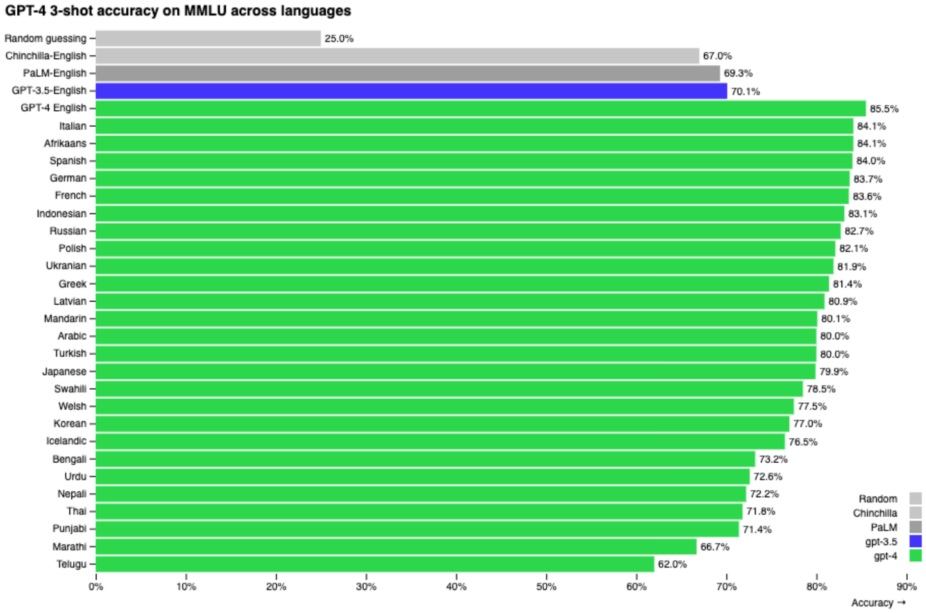


Figure 3.2: Performance of GPT-4 in different written languages. Taken from OpenAI[43]

3.3 Creating a system for metrics/rating of tasks

Our pre-master project investigated the relationship between the complexity of code snippets and the perceived difficulty of that code. Knowing there is relatively strong correlation between code complexity and perceived task difficulty, measuring the code complexity of tasks generated by our system could provide valuable data which can be used to :

- estimate the perceived difficulty of a programming task.
- Fine-tune weights of the rating system to more accurately analyze variants created by the generation system.
- Provide metrics which can be used to provide beginner students with beginner tasks, intermediate students with intermediate tasks etc.
- Gain a deeper understanding of what makes a programming task difficult.

As mentioned in section 2.3.1, Kasto et al.[12] chose the following intrinsic software metrics to measure the difficulty of novice code tracing and "Explain in plain English" tasks.

- Number of statements (originally number of lines)
- Number of operands
- Cyclomatic Complexity
- Average nested block depth
- Average number of parameters
- Number of function calls
- Deepest nesting

Taking inspiration, we chose to use many of these metrics in our own evaluation system. Reviewing earlier CS1 programming tasks in the light of the metrics above, we believed many of them would correlate to the inherent difficulty level of the tasks as perceived by us. The following paragraphs outline our reasoning for making use of the specific metrics.

Tasks where the resulting code contains a larger **number of lines**, intuitively correlates to more complex requirements in the programming task. This is mainly because the number of lines is indicative of the scope and scale of the problem to be addressed. More lines in a solution imply a large problem space or intricate solutions.

Tasks where the **number of operands** is larger generally indicates more complex data manipulations and operations from input to output. This may be due to a higher number of variables, complex logical or arithmetic operations, or the use of advanced data structures.

A deeper **average nested block depth** in a task is typically a sign of more intricate control flow. This could include more nested loops, conditionals and/or other control structures, which necessitate a higher level of programming skill to implement correctly and efficiently.

Similarly, tasks with a higher **number of parameters** often denote a more complex program structure. This could involve more function definitions, more complex function calls, or a higher degree of interaction between different parts of the code. Each of these scenarios adds a layer of complexity to the task, requiring the programmer to manage and coordinate more variables and data flows.

When a task encompasses a higher **number of function calls**, it is likely that the task involves more elaborate functionality and algorithmic complexity. The calling of built-in functions will also be registered by this metric. The understanding and application of function calls generally imply a more advanced level of programming, and in turn a more complex task.

The following paragraphs contains point-by-point descriptions of the software metrics, and how we tweaked them to better fit our use case.

Number of lines

Initially, the measure "number of statements" was the total number of lines. However, as the report dealt with the Java coding language and its peculiar coding conventions, single opening or closing braces and brackets tend to take up whole lines. Therefore, they used the "number of statements" as an approximation of the number of effective lines.

We chose to use the metric "number of lines" due to Python's concise syntax. The way to measure this is simply to count the number of lines of code in a solution. In general, for the solution to a coding task to be more complex, the amount of code increases, and in turn the number of lines in a solution increases.

Number of operands

A variable or a value involved in an operation is called an operand. An operand is any object capable of being manipulated. The number of operands serves as an approximate measure of how many "moving parts" are in a solution. A higher number of operands suggests that the solution is more complex than a solution with a small number of operands. A higher number of operands also points towards the solution needing more mathematical calculations in order to provide the correct answer, which in turn makes it more comprehensive.

Initially, two methods were considered for measuring the number of operands in a code snippet. The first method involves parsing all words in a code snippet and filtering out the words not associated with built-in functions. The main problem with this method is the sheer number of built-in methods python provides. If using string matching, there is also a risk of incorrectly identifying i.e. a string method as an operand. This becomes even more apparent if you consider the number of string methods, list methods, dict methods, etc.

The second method was deemed better, which involves counting the number of operators instead of operands. Knowing the number of unary/binary/ternary operators and multiplying by the respective number, we get a good approximation of the number of operands with a fraction of the complexity.

Cyclomatic complexity

McCabe's cyclomatic complexity is a software metric used to indicate the complexity of a program by measuring the number of linearly independent paths through the source code of a program. Having a large amount of branching in a code snippet increases the skill required to avoid logic errors and bugs. The increased branching also makes the code harder to trace and understand in the context of tracing assignments.

There are existing tools such as RADON[50] made to calculate cyclomatic complexity in Python code, but in order to reduce dependencies we implemented our own measure. Our measure calculates the number of independent paths and decision points that exist in a code snippet by counting the number of branching statements.

Before implementing the cyclomatic complexity measure, we used "deepest indent" as a dumbed down cyclomatic complexity measure. This worked well as Python uses indentation to define blocks of code, rather than using braces or similar syntax found in other popular programming languages.

Average nested block depth

Average nested block depth measures the average depth of nested control flow blocks within a code snippet, a control flow block being a section of code enclosed by conditional statements or loops. The depth of a block is the number of enclosing blocks in which it is nested. This measure correlates with cyclomatic complexity, as a higher level of nested control flow structures indicates a higher average nested block depth. However, a high average nested block depth indicates that much of the code is executed within branching statements, increasing complexity.

Our measure calculates the average number of indents of all lines in a code snippet, ignoring comments and newlines.

Number of parameters

Parameters are the variables or inputs that a function or method requires to execute. The "Number of parameters" metric tracks the number of (non-unique) parameters in both custom functions and built-in functions.

When combined, these metrics should provide an accurate reflection of a task's complexity. Furthermore, by developing a complexity analysis program, evaluating the complexity of already existing introductory CS programming tasks and using those as a basis to evaluate the complexity of our own generated tasks becomes feasible.

Dataset gathering and making ratings comparable

When comparing absolute values returned from the metrics, two tasks can have a significant gap between their values, and the instructor cannot easily draw any information from this difference. For this reason, we implemented a normalized score ranging from 1 to 10. The normalized scores work the same for all metrics. A task returns a deterministic number when testing a metric, which is then multiplied by a weight to normalize its value, with the final score maxing out at 10. Using the "number of lines" as an example, with a weight of $0,33$ a task with 16 lines of code in its solution will be given a score of $0,33 \cdot 16 = 5,0$. A task with 100 lines of code will "max out" and give a weighted "number of lines" score of 10.

In the testing phase of the rating system, we required a dataset of programming tasks. Our preliminary dataset consisted of 20 tasks derived from w3resource[51], and were categorized into the buckets "easy", "medium" or "hard". One our rating system was fully implemented, a larger dataset was necessary for fine-tuning the weights within the system. Consequently, we leveraged solutions from the entirety of the TDT4110 Information Technology - Introduction course at NTNU from 2019. For consistency and simplicity, we chose "easy", "medium" and "hard" as the buckets for these tasks as well. The resulting data set consisted of over 130 task solutions.

The calibration of the rating system consisted of adjusting weights in order to map the different complexity metrics to a predefined value scale. The value scale we chose was to fit the "easy" tasks within the value interval 0-3, "medium" into 4-6 and "hard" fit into 7-10. The calibration process was semi-automated, utilizing a custom-built analysis tool to assess key data derived from each category of tasks. This data provided insights allowing us to manually modify the weights of the metrics in the system to achieve what we believed to be an accurate and fair representation of task complexity.

3.4 Evaluation method

In this section, the selected method of evaluating the artefact is discussed. Development of software aimed at a specific group of users benefits from prototype testing and assessment by the target end users. For this project, the targeted users are teaching staff, professors and student assistants responsible for building programming exercise sets. We define the mentioned group of users as exercise-authors. Equally important is the group of people whom the exercises are assigned to, namely students in the relevant programming course. The generation software needs both evaluation in terms of efficiency in use, but also in regards to the exercise sets it produces. Thus, the evaluation is divided into two sections: teaching staff evaluation, and student evaluation. The selected format for both evaluations is real-world scenario user testing with a following interview for registering feedback.

3.4.1 Student demonstration

To evaluate exercises generated by the artifact, testing on the target group is vital. To organize a demonstration and testing on students, generated exercises should be provided in a realistic setting. A sufficient number of diverse students need to be selected. The student demonstration will also allow us to gain insight into the perceived quality of the exercises generated. In order to achieve an unbiased comparison of the automatically generated exercises and the manually created exercises, the students should be unaware how their respective exercise sets are produced.

3.4.2 Teaching staff demonstration

User testing is a crucial step in assessing the effectiveness and viability of software systems, particularly when tailored to specific user groups. In the context of automated programming task generation software, conducting user testing with task authors, including teaching staff and professors, holds immense significance. This subsection explores the rationale behind user testing on task authors and elucidates the benefits it brings to assessing the software's efficiency and utility. Additionally, it outlines the recommended approach for conducting user testing with task authors.

Software Validation in Real-World Scenarios

By involving task authors, such as teaching staff and professors, in the user testing process, the software is subjected to real world scenarios and authentic use cases. Task authors possess extensive experience in creating programming tasks, coupled with a deep understanding of the nuances and complexities involved. Their expertise allows them to provide valuable insights and comprehensive feedback, ensuring the software's validation under authentic conditions. User testing with task authors enables the evaluation of how well the software aligns with their existing workflows and requirements.

Assessment of Time-Saving Potential

One of the primary objectives of the automated programming task generation software is to save significant time for the task authors. User testing provides a unique opportunity to measure the software's time-saving potential in a controlled environment. Through direct observation and feedback, task authors can gauge the efficiency of the software in generating a variety of programming tasks. By comparing the time required to create tasks manually with the time taken using the software, the extent of time savings can be quantified.

Identification of Usability Issues and Improvements

User testing with task authors allows for the identification of usability issues and potential improvements within the software. Task authors can assess the user interface, interaction design, and overall user experience to provide constructive feedback. By analyzing their observations and suggestions, developers can identify pain points, streamline the user interface, and enhance the software's usability. Task authors' feedback also helps in identifying any limitations or shortcomings of the software, enabling targeted improvements to further optimize its utility and identify further use-cases for task authors.

Approach for User Testing with Task Authors

To ensure a comprehensive and effective user testing process with task authors, a structured approach should be followed. The recommended approach includes the following steps:

Participant selection: Identify a diverse group of task authors, including teaching staff and professors, who have experience in creating programming tasks.

User testing sessions: Conduct individual or group user testing sessions where task authors interact with the software and perform tasks. Encourage participants to think aloud, articulating their thoughts, impressions, and any difficulties encountered during the process.

Chapter 4

Related Work

This chapter consists of a review of literature related to our thesis, seeking to contextualize our study within the field. The section seeks to review related work which has served as a foundation to build from and upon. and highlights previous work our thesis builds upon, and where our study fits within the existing body of knowledge. The section focuses on the automatic generation of programming tasks, the impact of LLMs on introductory programming, and its results.

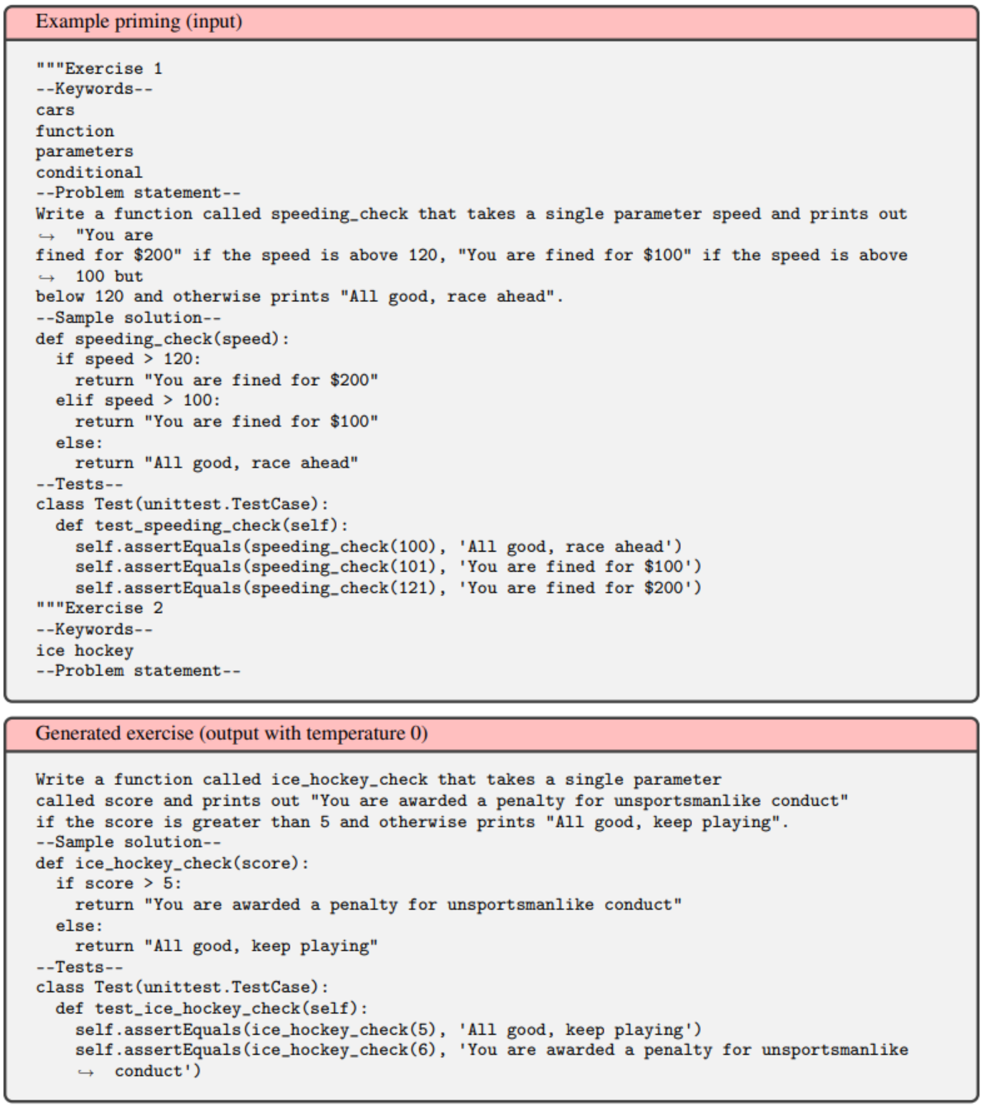
4.1 Automatic Generation of Programming Exercises and Code Explanations using Large Language Models

Sarsa et al. wrote the paper "Automatic Generation of Programming Exercises and Code Explanations using Large Language Models"[52], focusing on the utilization of the large language model OpenAI Codex[40] to create two key learning resources meant for programming courses: Programming exercises, and code explanations. The paper evaluates the quality and novelty of the exercises and explanations created by the AI model, both qualitatively and quantitatively.

The paper discusses theory regarding the value of practice and feedback in introductory programming courses, and explains how code explanation abilities are closely related to code understanding. They proceed to examine how machine learning models fare in the realm of code generation, highlighting the success of recent similar papers.

To generate programming exercises and code explanations, they provided Codex with docstrings containing a manually created problem description, sample solution, and test cases. A "second" exercise was started, but was not completed. Codex is meant to understand the first task as a form of blueprint, and complete the second exercise mirroring the structure of the provided exercise. An example of input and output can be seen in figure 4.1.

They also experimented with adding programming concepts("conditionals",



Example priming (input)

```
"""Exercise 1
--Keywords--
cars
function
parameters
conditional
--Problem statement--
Write a function called speeding_check that takes a single parameter speed and prints out
↪  "You are
fined for $200" if the speed is above 120, "You are fined for $100" if the speed is above
↪  100 but
below 120 and otherwise prints "All good, race ahead".
--Sample solution--
def speeding_check(speed):
  if speed > 120:
    return "You are fined for $200"
  elif speed > 100:
    return "You are fined for $100"
  else:
    return "All good, race ahead"
--Tests--
class Test(unittest.TestCase):
  def test_speeding_check(self):
    self.assertEquals(speeding_check(100), 'All good, race ahead')
    self.assertEquals(speeding_check(101), 'You are fined for $100')
    self.assertEquals(speeding_check(121), 'You are fined for $200')
"""Exercise 2
--Keywords--
ice hockey
--Problem statement--
```

Generated exercise (output with temperature 0)

```
Write a function called ice_hockey_check that takes a single parameter
called score and prints out "You are awarded a penalty for unsportsmanlike conduct"
if the score is greater than 5 and otherwise prints "All good, keep playing".
--Sample solution--
def ice_hockey_check(score):
  if score > 5:
    return "You are awarded a penalty for unsportsmanlike conduct"
  else:
    return "All good, keep playing"
--Tests--
class Test(unittest.TestCase):
  def test_ice_hockey_check(self):
    self.assertEquals(ice_hockey_check(5), 'All good, keep playing')
    self.assertEquals(ice_hockey_check(6), 'You are awarded a penalty for unsportsmanlike
    ↪  conduct')
```

Figure 4.1: Taken from Sarsa et al.[52], Appendix C. The figure shows an example priming used on OpenAI Codex to generate a programming task. The priming contains a detailed description of an exercise named "Exercise 1", including keywords, the problem statement, a sample solution, and a test set. The prompt then asks for an "Exercise 2", in where Codex generates the exercise seen at the bottom of the figure. The goal is for Codex to analyze and gather inspiration from the exercise example provided, and generate an exersize with identical structure but different contents. Note also that the only prompt given to Codex, is the Keyword "ice hockey". Based on only this one keyword, Codex is able to generate a sound exercise with the overarching theme of ice hockey, with a sample solution and a series of tests. Note that this is one of the instances in which everything worked as intended, which was not the case most of the time(See "Threats to validity"[52] (p. 37-38)).

"loops", etc) and contextual concepts("hiking", "fishing", etc.) to the priming, but found that the prompts not containing explicit concepts were also meaningful. Using different configurations of their contextual concepts, programming-related concept sets and exercise primings, the two different temperatures of 0 and 0,75, and generating two exercises, they made a total of 240 programming exercises used to evaluate the performance of Codex. Temperature is a parameter that controls the randomness in the model's output on a scale from 0 to 1, with a higher score meaning more "randomness", and in turn more variance.

They proceeded to manually assess the a randomly selected set consisting of half of the generated exercises, using the assessment table shown in figure 4.2, with the results shown in figure 4.3.

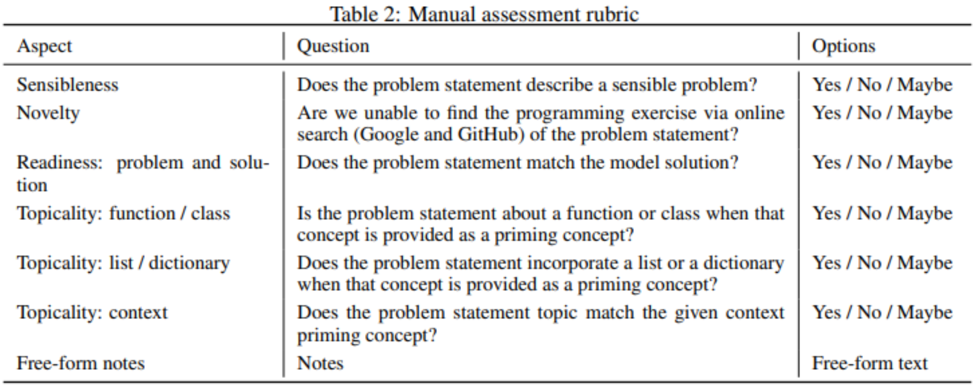
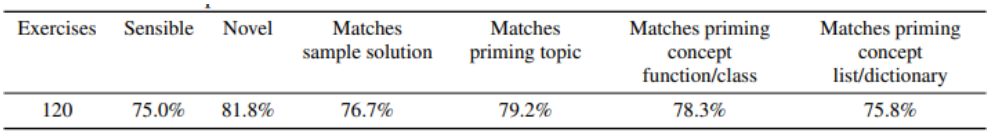


Table 2: Manual assessment rubric

Aspect	Question	Options
Sensibleness	Does the problem statement describe a sensible problem?	Yes / No / Maybe
Novelty	Are we unable to find the programming exercise via online search (Google and GitHub) of the problem statement?	Yes / No / Maybe
Readiness: problem and solution	Does the problem statement match the model solution?	Yes / No / Maybe
Topicality: function / class	Is the problem statement about a function or class when that concept is provided as a priming concept?	Yes / No / Maybe
Topicality: list / dictionary	Does the problem statement incorporate a list or a dictionary when that concept is provided as a priming concept?	Yes / No / Maybe
Topicality: context	Does the problem statement topic match the given context priming concept?	Yes / No / Maybe
Free-form notes	Notes	Free-form text

Figure 4.2: Table taken from Sarsa et al.[52], page 8. The table is the manual assessment rubric used to evaluate the performance of Codex, and the usability and utility provided by the generated tasks.



Exercises	Sensible	Novel	Matches sample solution	Matches priming topic	Matches priming concept function/class	Matches priming concept list/dictionary
120	75.0%	81.8%	76.7%	79.2%	78.3%	75.8%

Figure 4.3: Table taken from Sarsa et al.[52], page 9. The table shows a summary of the manualyl evaluated programming exercises.

The results showed that 75.0% of the tasks were deemed "sensible", meaning that the requirements given in the tasks were described clearly within a logical context. 81.8% of the tasks were deemed "novel", meaning that when using the exercise description as a query to search Google or GitHub, the search returned no valid results. "Matches sample solution" indicates that the generated solution matches the textual description.

In addition to these results, they found that 84, 6% of the generated exercises had a sample solution, and that 89, 7% of the exercises that had a sample solution, had solutions that ran without errors. Only $70,8\%$ of tasks had tests, and of these only $30,9\%$ of tasks had tests that all passed.

4.1.1 Summary

Even though the results are promising, the success rates presented above makes the unsupervised generations of programming tasks quite unreliable. The fact that only 70% of tasks had accompanying tests, of which only worked 30% of the times, emphasize the important point that language models doesn't always do what they're told. If one were to implement automatic test generation in a system, larger and more reliable language models are a requirement.

The model they used for task generation is already considered deprecated. The article uses $code-davinci-001$, but as of June 2023, several better models such as $GPT-4$, $code-davinci-003$, $text-davinci-003$ exist. These newer models are considerably better at interpreting the natural language prompts given than the earlier iteration models. Some of the newer models may also fit the given use-case better. The article also points to many sensible precautions one must make before utilizing a similar system.

To summarize the paper, the results suggest that OpenAI Codex can generate both sensible and sometimes even ready-to-use programming exercises and explanations. They found that providing the model with a combination of keywords related to programming concepts and themes, resulted in the generation of sample solutions, test cases and detailed code explanations.

Especially the introductory line in the "Future Work" section is interesting, which is quoted here: "We see great potential for future work utilizing Codex and other similar models in the context of programming education. Given the positive results we have observed in terms of programming exercise generation, we are interested in developing an automated exercise generator powered by Codex that could be used by instructors. ..." [52](p. 11-12).

4.2 The impact of OpenAI Codex on Introductory Programming

Finnie-Ansley et al.[53] wrote the paper "The impact of OpenAI Codex on Introductory Programming" in 2022, where they systematically measure Codex's performance on introductory programming tasks and compare it to CS1 students. They also explore the variety that exists in the solutions generated.

The paper starts by asserting the fact that "a comprehensive review of introductory programming literature from 2018 makes no mention of tools that use AI to produce code, ..."[53](p. 11). In the same way as Sarsa et al.[52]'s paper, they go through the capabilities of the newest large language models and the solutions that are derivatives of them, such as OpenAI's GPT-3 and Codex[40].

They use two separate evaluations to assess the accuracy of Codex. The first assessment consists of prompting Codex with 23 programming questions previously utilized towards students at the authors institutions in 2020. The problem statements was used in assessments throughout a programming course, with Test 1 taking place in the middle of the course and consisting of 11 questions, and Test 2 taking place towards the end of the course and consisting of 12 questions. The results from Codex can be seen in figure 4.4.

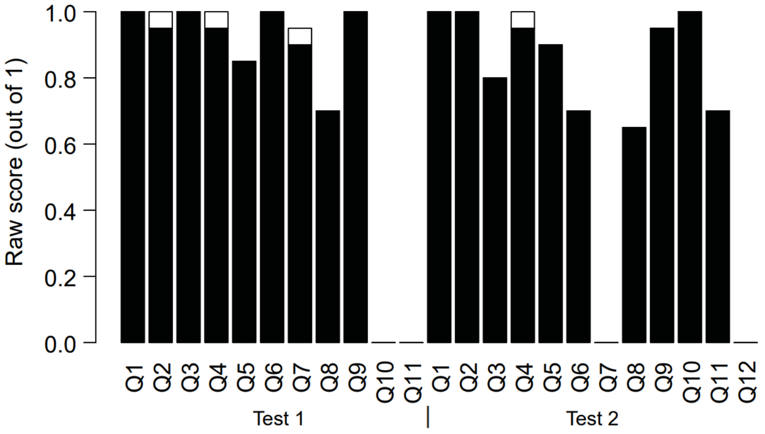


Figure 4.4: Taken from Finnie-Ansley et al.[53](p. 13). The figure shows the scores achieved by Codex in each of the 23 course problems. The lack of points seen in Test 1 Q10 and Q11, and Test 2 Q12 was a direct result of a trivial formatting error. Such errors in the generated code could easily be spotted and fixed by an instructor, which the authors point out would result in Codex recieving full marks in the mentioned tasks.

The tasks were fed to Codex in the same way they were provided to students. Using the same marking scheme as for the students, Codex's responses scored a total of 15.7/20(78.5%) in Test 1 and 19.5/25(78%) in Test 2. The study viewed Codex as a student, in which it came in at 17th place when compared to the 71 students. Student scores can be seen in figure 4.5.

The second Codex assessment, was completed by providing Codex with seven variations of the classic Rainfall Problem[54], and having it solve them. No example test cases were provided as part of the problem. Each problem description was provided to Codex 50 times, resulting in 350 responses. Each of the responses were ran through 10 test cases, which resulted in a total of 3500 evaluations. Table 4.1 illustrates the marked results of Codex, from 0(minimum score) to 1(max score).

The results were varying, with Codex outperforming the 149 students in Si-

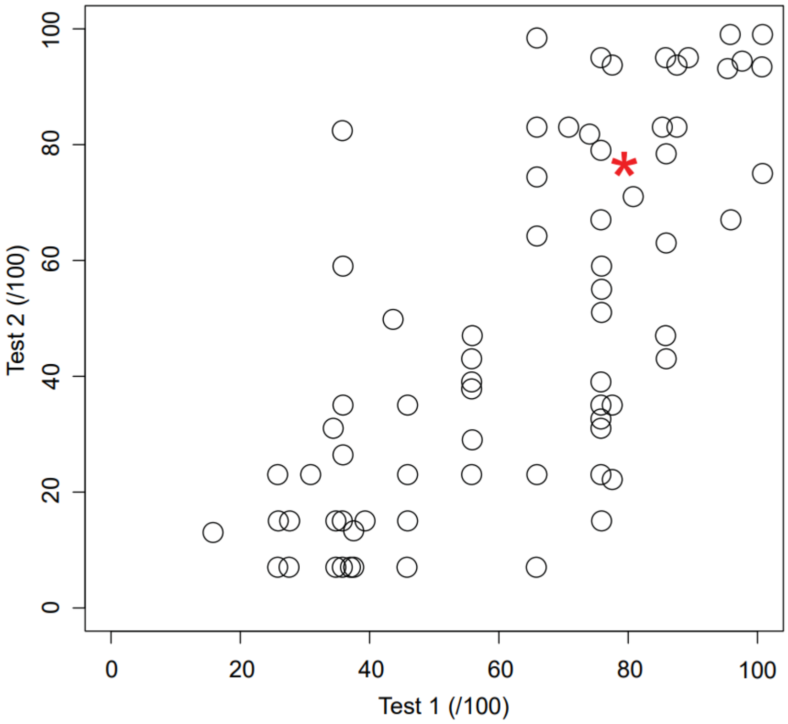


Figure 4.5: Figure taken from Finnie-Ansley et al.[53]. The axes shows test scores 1-100, where Codex's scores are market with a red asterisk.

Variant	Mean	Median	Max Score	Stddev
Soloway[54]	0.63	0.90	1.00	0.40
Simon[55]	0.48	0.50	1.00	0.28
Fisler[56]	0.61	0.70	1.00	0.26
Ebrahimi[57]	0.19	0.05	1.00	0.26
Guzdial et al. [58]	0.47	0.30	1.00	0.22
Lakanen et al. [59]	0.44	0.70	0.90	0.32
apples	0.54	0.60	1.00	0.34

Table 4.1: Table taken from Finnie-Ansley et al.[53](p. 15). The column "Variants" contain the authors of the different versions of the rainfall problem. The full wording variations can be found on page 14 of the paper by Finnie-Ansley et al.[53].*Apples* is the authors own version of the rainfall problem which uses harvested apples instead of rainfall as the contextual setting, hence the name. Each response was graded against the test case, with each test case contributing 0.1 to the overall score.

mon[55], where the average partial score was 29%. Codex performed similar to the 120 students in Guzdial[58] that had the average partial score of 46%, and performed worse than the 139 students in Lakanen et al. [59], which boasts an average partial score of 69%. To summarize, it is clear that OpenAIs Codex is able to solve programming tasks with impressive accuracy. As seen in figure 4.4, its performance is sometimes hindered due to trivial formatting errors or similar, resulting in the need for human interaction to solve the problem.

Chapter 5

Result

The development and assessment of the software in this project was guided by agile development, and separated into three sprints with durations of four weeks. It became natural to divide the artifact into three main parts, corresponding to the following sprints:

- Sprint 1: Variant task generation system
- Sprint 2: Rating system and exercise builder
- Sprint 3: REST API and frontend

The two main segments of the backend are the variant task generation and the rating system, making up the structure that automatically generates and rates variant tasks using GPT-based models. The REST API and frontend, on the other hand, are built on top of the mentioned systems, to facilitate effective use of the system by teaching staff. Instead of a strictly iterative process, where all the mentioned parts are improved during sprints, it was decided to divide the development into the three chunks. This decision was taken due to the limited developer workforce of two people, where the ability to delegate is limited. A negative effect of this approach was that assessment of the prototype would not be conducted before Sprint 3, as it required the complete system to implement a demonstration.

This chapter is structured according to the process of implementing and assessing the artifact, leading to the results being presented chronologically. Firstly, the software developed is described and the resulting system is explained, sprint by sprint. Some details not deemed relevant are left out, while others are considered more carefully. This prioritization is done to highlight the most crucial concepts applied to achieve efficient variant task generation and rating. Further down, the user-experience and design of the system is presented as utilized by teaching staff. This serves as a review of using the artifact before attending to the demonstration and assessment on real users. The chapter is concluded by examining the results from the assessments, both the student demo and teaching staff interviews.

5.1 Variant task generation system implementation

This section focuses on the implementation of the project artifact, which serves as an extension of the prototype developed during the pre-project phase. The aim of the section is addressing Research Question 1[1.2], by showcasing one implementation of a system that allows creating programming exercises using GPT-models. It is important to note that the software is still in its prototype stage. It has been given the working title "TaskMaster" and encompasses task-generation python methods, a REST API server, and a frontend interface for task authors.

The primary functionality of TaskMaster revolves around the generation of numerous variant tasks based on concise textual descriptions. These variant tasks can then be combined to form exercise sets. The backend of TaskMaster consists of the Python classes and methods responsible for exercise generation, along with the REST API server. The backend follows a module-based approach and is constructed using the object-oriented paradigm. The JavaScript-code implemented as a React-app[60] is referred to as the frontend, and makes up the graphical user interface.

5.1.1 Defining important keywords in the system

- **Variant template:** The variant templates are database entities describing the content of the desired variants to be generated, dictating the GPT queries used. The templates make up the common recipe for the unique variant tasks generated by the system. One example of a variant template is seen in Figure 5.1.
- **Variant task:** A variant or variant task is one of many related tasks generated by the same template. They are called variants because the underlying concepts are the same, but small differences make sure they are not identical. The variant type is represented with the following properties:
    - *Code:* The main element of the variant. Needs to be a python function.
    - *Description:* Describes the code from a task perspective. This means provididing hints, but not disclosing too much information.
    - *Callable:* Name of the function that can be called, with randomized parameters.
    - *Output:* The output of executing the code, used both to verify that the function is running, and as examples in the task description.
    - *Distractors:* A list containing one key line in the code, alongside three incorrect alternatives to the line.
    - *Rating:* Normalized ratings of the code complexity, including nesting, parameters, operands, numbers of lines and cyclomatic complexity.
    - *Meta-data:* The GPT-model used, timestamp, etc.
- **Batch of variant tasks:** Variants are built in batches of more than one. The number is decided by the task author.

- **Task:** The term is often used interchangeably with variant tasks, and refers to either a composition of the key elements in a variant, or a complete variant task formatted on one of the supported task formats.
- **Task author:** The intended user of the system with the goal of producing variant tasks and possibly combining these into exercise sets.
- **Exercise set:** A combination of formatted tasks. Exercise sets are generated in the plural, with unique variants of each corresponding task.
- **GPT query:** Queries are sentences containing instructions to guide the output of a GPT-prompt.
- **GPT prompt/request:** Prompts or requests to a GPT-model contains a query, and generates a response. Prompts typically take 10-20 seconds to complete through the OpenAI API.

Code listing 5.1: Example of a variant template

```
{
    "_id" : ObjectId("64149b427af25824d3224814"),
    "Name" : "Sum odd numbers",
    "Description" : "sums all odd numbers in a list",
    "IncludeLoops" : true
}
```

5.1.2 Sprint 1: Variant task generation system

The first sprint included setting up all the core elements of the task generation. Implementing a ‘OpenAIClient‘ class and make use of GPT-prompts to build variant tasks according to the type defined in 5.1.1 was prioritized. The most substantial part of this sprint was therefore developing a ‘VariantGenerator‘ class.

OpenAI API communication - ‘OpenAIClient‘ class

The handling requests to OpenAI is done using a class named “OpenAIClient”. This approach abstracts and simplifies the process of making API calls, and is more structured than making requests directly from all methods dependant on the API. The OpenAIClient imports the “openai” python package. As the class is instanced as a singleton in the backend, it is first instantiated with a specific GPT model. The GPT model can then be changed later, using a getter-method. A list of available models is fetched in a method named “get_models()”. The main method is “gpt_request(prompt: string, history: list, temperature: float)”, handling requests to the OpenAI API. Depending on the chosen GPT model, various methods from the OpenAI package are employed. Recognizing that requests to the OpenAI API may occasionally time out or fail, a persistent solution has been implemented to automatically retry requests up to three times. The OpenAIClient also maintains a record of the number of requests per batch of generated variants.

Caching prompts and responses

The decision was made to incorporate a caching system within the OpenAIClient for several reasons. Firstly, each request made to the OpenAI API incurs a small monetary cost that accumulates over time. Additionally, the time required for each request is significant, ranging from 10 to 30 seconds. Extensive testing revealed that responses often remained identical when provided with the same prompt and temperature settings. While the option of storing each response in a database was considered, it was ultimately deemed unnecessary for other system functionalities. Instead, the decision was made to implement caching using a local file. When the constructor is called, the OpenAIClient reads the caching file into memory, and after completing requests, it writes the updated cache back to the file. This approach has proven to be highly effective during user testing.

Interpret the response from the GPT - ‘CodeDeserializer‘ class

Responses from the OpenAI API are a combination of code and text, formatted as a string. One example can be seen in Listing 5.2. The structure of the response varies between prompts. It is therefore necessary to include a deserializer that identifies the relevant code and separates functions if more than one is provided. A class “CodeDeserializer” was built for this purpose. It functions as a singleton, and contains the method “deserialize(response_string: string)” which returns a list of python functions.

Code listing 5.2: Example of response from GPT-request

```
Prompt:  Build  this python-code: A function that reverses a list using a loop
Response:
Here’s a Python function that reverses a list using a loop:

‘‘‘python
def reverse_list(lst):
    new_lst = []
    for i in range(len(lst)-1, -1, -1):
        new_lst.append(lst[i])
    return new_lst
‘‘‘

This function takes a list as an input and creates a new list that contains the
elements in reverse order. It does this by iterating over the indices of the
original list backwards (from the last index to the first), and adding each
element to the new list in reverse order. Finally, it returns the new list.

Here’s an example of how to use the function:

‘‘‘python
my_lst = [1, 2, 3, 4, 5]
reversed_lst = reverse_list(my_lst)
print(reversed_lst) # Output: [5, 4, 3, 2, 1]
‘‘‘
```

Generation using variant templates - ‘VariantGenerator‘ class

The core of the backend is the “VariantGenerator” class, responsible for generating any requested number of variants based on a variant template. These variants include the code, example of execution, evaluation of code complexity, and a task description. The VariantGenerator is instantiated with a variant template that outlines the specifications for producing the variant tasks. To start the process, the method “generate_code(amount: int)” determines the desired number of variants to request at different temperature settings, which dictates the levels of randomness. Queries are then constructed accordingly and used to call the “gpt_request”-method within the OpenAIClient for each query. The resulting responses are aggregated into a list, with any identical functions being filtered out.

After obtaining the code, which serves as the foundation for each variant, the VariantGenerator proceeds by requesting an appropriate task description for each function. This is achieved by utilizing the "gpt_request" method once again. Previous requests and responses are stored and provided as context for subsequent requests. Requested task descriptions are not intended as complete explanations of the code but rather as descriptions of what the function should do, occasionally including hints. One example of the generated code and task description in a variant is seen in Listing 5.3.

Code listing 5.3: The code and task-description of a generated variant

```
def reverse_list(lst):
    reversed_list = []
    index = len(lst) - 1
    while index >= 0:
        reversed_list.append(lst[index])
        index -= 1
    return reversed_list
```

```
Skriv en funksjon som tar inn en liste som argument og returnerer en ny liste
der elementene er i motsatt rekkefølge av den opprinnelige listen. Du skal
bruke en while-løkke for å iterere gjennom elementene i den opprinnelige listen
ved å starte fra slutten av listen og legge dem til i den nye listen. Du kan
ikke bruke innebygde funksjoner som ‘reversed()‘ eller liste-slicing for å løse
denne oppgaven.
```

Executing code in variants

The third step in generating a complete batch of variant task is identifying the function definition and creating randomized arguments to execute the code for each variant. The “get_callables” method in the VariantGenerator class considers the generated function of a variant, which is still represented as a string at this stage. It extracts the function name and a list of required arguments from the function’s docstring. For each argument, methods in the “VariableFactory” class are used to provide values of the correct data type. Subsequently, the "run_code" method spawns a new process and attempts to execute the function using the built-in "exec()" function. Multiprocessing is employed to handle potential issues arising from functions with endless loops or similar complications. These issues arose during the development phase, as the code is solely provided by a GPT model and thus its reliability is not absolute. Therefore, if the execution exceeds five seconds, the process is terminated. This approach has proven to successfully mitigate the issue.

Code listing 5.4: The code and example of execution of a generated variant

```python
def reverse_list(lst):
    """
    Reverses a list using a while loop.
    Parameters:
    lst (list): The list to be reversed.
    Returns:
    list: The reversed list.
    """
    reversed_lst = []
    i = len(lst) - 1
    while i >= 0:
        reversed_lst.append(lst[i])
        i -= 1
    return reversed_lst


print(reverse_list([86, 99, 32, 79, 38]))
>>> [38, 79, 32, 99, 86]
```

One example of a function and the generated function call is seen in Listing 5.4. The execution of code serves two primary purposes. Firstly, it facilitates the detection of erroneous and void functions. Variants containing functions that do not run are excluded from further consideration. Secondly, code execution and its corresponding output provide valuable contributions to the task description by serving as an illustrative example. In order to capture the output generated by print statements within the functions, the standard output stream is redirected to a variable during code execution. It is worth noting that certain functions may produce an extensive number of output lines, for example printing every integer between 0 and 100. To prevent task descriptions from becoming excessively lengthy in these cases, only the first five and last five output lines are retained.

Determining the type of arguments to call functions - ‘VariableFactory‘ class

Examples of function execution are often included directly in the responses to GPT requests, alongside the corresponding code. This is seen in Listing 5.2. However, due to this not always being the case, in addition to the inconsistent format of these examples, it was decided to instead use the proprietary code execution approach. As previously mentioned, the implementation relies on a VariableFactory class, which is responsible for providing suitable argument values for function calls. While argument names are identified straightforwardly, determining their types can be more challenging. Consider the two examples of function definitions:

“def find_max(lst):” and “def foo_foo(bar):”. In the first case, the argument type is intuitively clear, whereas in the second case, it is more ambiguous.

Two techniques are used in order to build randomized argument values. Firstly, the code is examined for explicit typing hints or type annotations within the function’s docstring. A docstring in a generated function is seen in Listing 5.4, while the same function typed is shown in Listing 5.5. The VariableFactory is equipped to identify these specifications and extract the argument types accordingly. In cases where the docstring and typing information are absent, the VariableFactory resorts to predefined lists of argument names. For instance, arguments named “list”, “lst”, “array”, “numbers” are identified as lists, while names such as “i”, “j”, “n”, “m” and “number” implies integers. It is important to note that this implementation is not flawless, and testing has revealed instances where the code execution within variants encounters arguments with unsuitable types. When generating argument values, the VariableFactory utilizes the Python "random" module to generate random values according to the identified type, in the “build_variable" method.

Code listing 5.5: The code of a variant with explicit typing

```python
def reverse_list(lst: List) -> List:
    reversed_lst = []
    i = len(lst) - 1
    while i >= 0:
        reversed_lst.append(lst[i])
        i -= 1
    return reversed_lst
```

5.1.3 Sprint 2: Rating system and exercise builder

In the second sprint, implementing the rating system devised in Section 3.3 was the primary goal. This entailed building a ‘VariantEvaluator‘ class along with helper-functions for each metric used. In addition, higher-order classes such as a ‘VariantFactory‘ and ‘ExerciseFactory‘ class were outlined in order to facilitate generating batches of variant tasks organized in exercise sets.

Evaluating complexity in generated variants - ‘VariantEvaluator‘ class

With the three main parts of a variant in place, the python function, code description and example of execution, the complexity of variants are evaluated according to the complexity metrics described in Section 2.3.1. Evaluation of code complexity is implemented in the “VariantEvaluator” class in the method “evaluate(variants)”. The evaluation is performed on each variant in a batch, and utilizes a helper function for each rating feature. These pure functions are external to the VariantEvaluator, and accept a string representation of code as their sole argument. The rating functions are “number_of_lines”, “number_of_operands”, “number_of_function_calls”, “deepest_nesting”, “cylomatic_complexity”, “average_nested_block_depth”, and "number_of_parameters", corresponding to the ratings shown in Figure 2.1.

Intrinsic software metric	Weight
Number of lines	0,33
Number of operands	0,35
Cyclomatic complexity	1,5
Average nested block depth	3
Average number of parameters	0,95
Number of function calls	0,3
Deepest nesting	1,9

Table 5.1: The table shows the intrinsic software metrics used to evaluate complexity of tasks, and the weights given to normalize the scores.

The results from the process of calibrating the rating system as described in section 3.3 are shown in table 5.1.

In addition to all the individual ratings, an average rating is computed, serving as an indicator of the overall complexity of each variant. To get an idea of the differences in complexity within a batch of variants, the standard deviation in regards to each respective rating is also calculated and displayed. Two examples of generated functions with resulting ratings are seen in Listings 5.6 and 5.7.

Code listing 5.6: The code and rating of a simpler variant

```python
def reverse_list(lst):
    reversed_lst = []
    for i in range(len(lst) - 1, -1, -1):
        reversed_lst.append(lst[i])
    return reversed_lst


Average: 2.74
Block-depth: 3
Cyclomatic: 3
Function-calls: 1.2
Lines: 2
Nesting: 3.8
Operands: 2.4
Parameters: 3.8
```

Code listing 5.7: The code and rating of a moderate variant

```python
def bubble_sort_optimized(lst):
    n = len(lst)
    while n > 1:
        swapped = False
        for i in range(n-1):
            if lst[i] > lst[i+1]:
                lst[i], lst[i+1] = lst[i+1], lst[i]
                swapped = True
        if not swapped:
```

```
            return lst
        n -= 1
    return lst
```

```
Average: 5.06
Block-depth: 6
Cyclomatic: 7.5
Function-calls: 0.9
Lines: 4.3
Nesting: 7.6
Operands: 6.3
Parameters: 2.8
```

Controlling the process of generating variant batches - ‘VariantFactory‘ class

The classes from the first sprint fulfill distinct roles within the variant generation process, aligning with the principles of modularization in software development. This design choice was made to facilitate future expansions of the variant system. However, as a consequence, these classes rely on a unifying class that governs the flow of data. This role is fulfilled by the "VariantFactory" class, which serves as the orchestrator for constructing batches of variant tasks through its "build_variants(amount)" method. In this context, the VariantFactory can be seen as a "director" following the object-oriented "builder" design pattern[47]. In addition to the variants themselves, a meta-object is returned, providing information about erroneous variants, duplicates, and timestamps.

Prior to the construction of variants, the desired variant template is set using the "initialize_template(template)" method, which in turn affects the other classes used by the VariantFactory. Upon completing the construction of a batch, the VariantFactory calls the "format_variant" method on each variant, ensuring that each variant is constructed according to the defined variant task type5.1.1.

Using variants to construct tasks on specific formats - ‘TaskFormatter‘ class

The variants built by the VariantFactory contain all the necessary components to construct exercises and programming tasks. However, these components need to be assembled according to the task formats as outlined in Section 2.2.1. To facilitate this process, the “TaskFormatter” class has been developed. The primary method “format_tasks(variants, format)” takes in a list of variants and returns tasks in the specified format, together with the solution to each task. The supported task formats are “code-writing”, “find-errors”, “fill-in-blanks”, “find-output”, “parson-problem” and “multiple-choice”. Each format is associated with a dedicated formatting method that takes a variant as input and generates a formatted task. One example of multiple-choice task formatted from a variant is seen in Listing 5.8.

Code listing 5.8: A variant formatted as a multiple-choice task, with the correct alternative labelled

```
# Alternative A (Correct, from variant code):
def sum_odd_numbers(numbers):
    sum_odd = 0
    for number in numbers:
        if number % 2 != 0:
            sum_odd += number
    return sum_odd

# Alternative B (Incorrect):
def sum_odd_numbers(numbers):
    sum_odd = 0
    for number in numbers:
        if number % 2 == 0:
            sum_odd += number
    return sum_odd

# Alternative C (Incorrect):
def sum_odd_numbers(numbers):
    sum_odd = 0
    for number in numbers:
        if number % 2 > 0:
            sum_odd += number
    return sum_odd

# Alternative D (Incorrect):
def sum_odd_numbers(numbers):
    sum_odd = 0
    for number in numbers:
        if number % 2 == 1:
            sum_odd += number
    return sum_odd
```

Using distractors in task formats

Although the task formatter could certainly be more versatile in terms of dynamically defining formats, it was deemed sufficient to support the six current predefined formats. Among these formats, three of them distinguish themselves in terms of implementation. While for instance the “code-writing”-format only provides the task description and example of execution directly from the variant, and “find-output”-tasks provide the code itself, other formats require more context. Specifically the “find-errors”, “fill-in-blanks”, and “multiple-choice”-formats all require distractors relating to the function of the variant. Distractors are provided by prompting a GPT model, because they require identifying crucial parts of the code. For instance, consider the following function:

```
def reverse_list(lst):
    reversed_lst = []
        for i in range(len(lst) - 1, -1, -1):
            reversed_lst.append(lst[i])
    return reversed_lst
```

Deciding which line to change and what distractor to use is intricate. Therefore GPT prompts are utilized, providing the code and querying for incorrect alternatives on one key code line. For the example above, the following list of code lines were returned, ideal when assembling the distractor formats:

```
Correct line: for i in range(len(lst) - 1, -1, -1):
Distractor #1 for i in range(0, len(lst), 1):
Distractor #2 for i in range(-1, len(lst) - 1, -1):
Distractor #3 for i in range(len(lst), 0, -1):
```

Controlling the process of generating exercise-sets - ‘ExerciseFactory‘ class

Similarly to how the VariantFactory directs the building batches of variants, the “ExerciseFactory” class is implemented to arrange the complete process of building variant tasks and formatting them into exercise sets. An exercise set is a composition of variant tasks. By building batches of N variants for M variant templates, the TaskFactory is able to assemble N unique exercise sets of M tasks. Along with each exercise set, a set of corresponding solutions is generated.

Randomized distribution of concepts
A crucial aspect of the exercise building is randomly shuffling the order variant tasks before sequentially assigning them to the requested exercise sets. This mechanism was implemented due to the tendency of the GPT model to consistently apply different programming concepts in the same order for each batch of variants. Specifically, the consequence could be that exercise set #1 contained an overweight of variant tasks with “for-loops”, exercise-set #2 focused heavily on “while-loops”, and so on. Although a more deliberate distribution of programming concepts would be desirable, the current implementation utilizes a randomized order to address this issue effectively.

5.1.4 Sprint 3: REST API and frontend

The final sprint covered the finalization of the system. Functions to export the generated tasks and exercises to different file-formats were implemented. The system was converted to a web application, by adding both an API and a frontend to the existing set of python-classes. The underlying concepts of generating variant tasks were therefore completed ahead of this sprint.

Exporting generated exercise sets to PDF and QTI

After formatting the tasks in terms of structure, the more technical aspect of file formatting is required to make the exercise sets available to the users. Two functions are developed for this purpose: “generate_pdf(exercises, name)” and “generate_qti(exercises, name)”. The former is the simpler of the two, utilizing HTML templates to present the variant tasks within each exercise set, exported as PDF files. Additionally, a solution set is generated on a separate PDF alongside each exercise set. This approach has proven effective, and has effortlessly provided physical copies of generated exercise sets for the student demo.

Potential for QTI-formatting
On the other hand, exporting to QTI requires a more complex and specific formatting of the exercise sets, which has not been completed during this project. However, the potential for future implementation of the QTI format would enable the use of the generated exercise sets in online assessment platforms, such as Inspera Assessment[61]. This would also facilitate task authors in manipulating, adding and removing variant tasks in each exercise set through other platforms.

Make generating variants and exercises available through a REST API

On top of the python-classes and functions that make up the backend, a REST API is implemented using the Flask[62] web-framework. The implementation of this API is contained within the "Server.py" file, where specific endpoints are defined to allow users to modify and add variant templates, generate variants, and create complete exercise sets. The server itself plays a minimal role in terms of logic and instead relies on abstraction by calling on the methods exposed by the “Variant-Factory” and “TaskFactory”-classes. As these methods often take a while to finish, and run asynchronously, the server keeps track of ongoing requests. For instance when generating a batch of variants, the details of this process is added to a list. A specific endpoint is responsible for providing the list of ongoing requests to the frontend.

To communicate with other services, the server instantiates the OpenAIClient-singleton, as well as the “MongoDBClient” class that enables managing the variant templates stored in a cloud-based MongoDB[63] database. The methods implemented in the MongoDBClient are “get_template”, “get_templates”, “add_template”, “update_template” and “delete_template”.

5.1.5 Summarizing TaskMaster

In this section, we have delved into the comprehensive implementation of TaskMaster. TaskMaster comprises variant task and exercise set generation, a REST API server, and a frontend interface for task authors. Throughout the section, we have discussed the key components, design principles, and functionalities that constitute the backend and frontend of TaskMaster. The system demonstrates an implementation of programming exercise generation using GPT-models, which was the main goal of Research Question 1(1.2). Research Question 2(1.2) relates to the challenges of building such a system, some of which have been presented in this section.

The backend of TaskMaster has been constructed using a module-based approach, implemented through object-oriented programming. The complete class-diagram is seen in Figure 5.1. The backend architecture incorporates a range

Figure 5.1: Structure of TaskMaster

of classes such as "OpenAIClient" for handling requests to OpenAI, "VariantGenerator" for generating variants, "VariantEvaluator" for evaluating code complexity, and "TaskFormatter" for formatting tasks according to different task formats. These classes work in tandem to ensure the smooth execution of task generation. The result is an experience more tailored towards task creation from start to finish than using the GPT-model interfaces such as ChatGPT[4] directly.

Complementing the backend, the frontend interface of TaskMaster is implemented as a React app. It serves as the graphical user interface, enabling task authors to interact with the system intuitively. The frontend provides a user-friendly environment for managing variant templates, and generating variants and exercise sets. A REST API server has been implemented to function as an intermediary between the frontend and the backend, providing endpoints that enable users to modify and add variant templates, generate variants, and generate complete exercise sets.

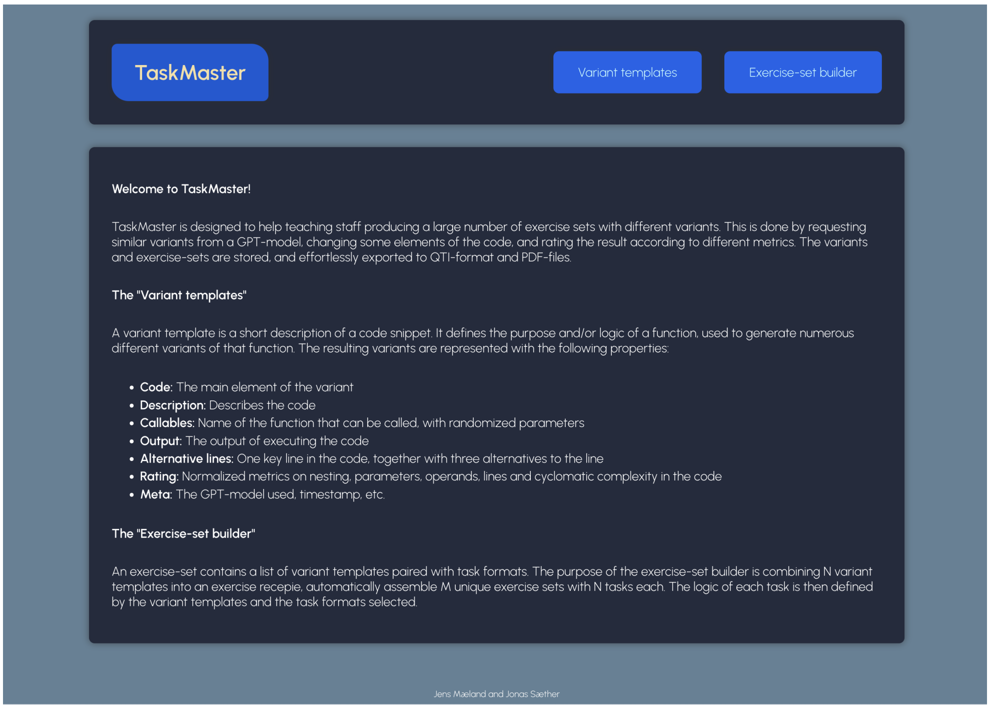
The source code is available on the GitLab repository[64]. In the README of the project, a link to the hosted version of the TaskMaster prototype is also accessible.

5.2 Operating the system from a user-perspective

The experience of using the artifact of the project is just as important as the underlying concepts. Users need to be able to effectively utilize the system. This section describes TaskMaster from the perspective of a user, and touches on Research Question 3[1.2] by discussing the use-cases and scenarios it is applied to.

5.2.1 Front page

The front page of TaskMaster contains a simple README-like section of text explaining the purpose of the two sub-pages, seen in Figure 5.2. This explanatory front page provides new users with a guide, and enables using the system effectively without a comprehensive demonstration. The "TaskMaster" logo on the top right always leads to the front page when clicked. The two buttons to the right are labeled "Variant templates" and "Exerise-set builder", corresponding to the two sub-pages. The development of the visual parts of the prototype were undertaken with a focus on inclusivity, specifically catering to individuals with visual impairments such as poor eyesight and colorblindness. It was also tested by one of the developers involved in this project, as they have deuteranopia (a form of color blindness) and reduced eyesight.



Welcome to TaskMaster!

TaskMaster is designed to help teaching staff producing a large number of exercise sets with different variants. This is done by requesting similar variants from a GPT-model, changing some elements of the code, and rating the result according to different metrics. The variants and exercise-sets are stored, and effortlessly exported to QTI-format and PDF-files.

The "Variant templates"

A variant template is a short description of a code snippet. It defines the purpose and/or logic of a function, used to generate numerous different variants of that function. The resulting variants are represented with the following properties:

- **Code:** The main element of the variant
- **Description:** Describes the code
- **Callables:** Name of the function that can be called, with randomized parameters
- **Output:** The output of executing the code
- **Alternative lines:** One key line in the code, together with three alternatives to the line
- **Rating:** Normalized metrics on nesting, parameters, operands, lines and cyclomatic complexity in the code
- **Meta:** The GPT-model used, timestamp, etc.

The "Exercise-set builder"

An exercise-set contains a list of variant templates paired with task formats. The purpose of the exercise-set builder is combining N variant templates into an exercise recepie, automatically assemble M unique exercise sets with N tasks each. The logic of each task is then defined by the variant templates and the task formats selected.

Jens Mæland and Jonas Sæther

Figure 5.2: Front page of TaskMaster, including a README

5.2.2 Variant templates list

The first of the two sub-pages is the "Variant template"-list, illustrated in Figure 5.3. Here, all variant templates stored in the database are listed, so that variants defined by one user can be accessed by others. Variant templates can be filtered using a search field. Adding a new template is done using the "Add" button. By clicking the button of one of the variant templates, that specific variant template will open.

5.2.3 Variant template editor

Upon selecting a variant template the menu on top will display the name of the selected template, along with a "Back"-button back to the list of templates. Two additional buttons enable copying and deleting the current template. In the center of the view, seen in Figure 5.4 the attributes of the template are displayed using editable fields. This is where users, or task authors, can edit the specifications of variant templates, influencing the variant tasks produced. Because of the stochasticity of generating variants, the user is provided with a "Testing ground" directly below the template editor. This facilitates testing the effect of adjusting existing templates or creating new templates straight away. When using the "Testing ground" to generate variants, the GPT-model to be used is selected using a dropdown menu, seen in Figure 5.5. The available models correspond to those

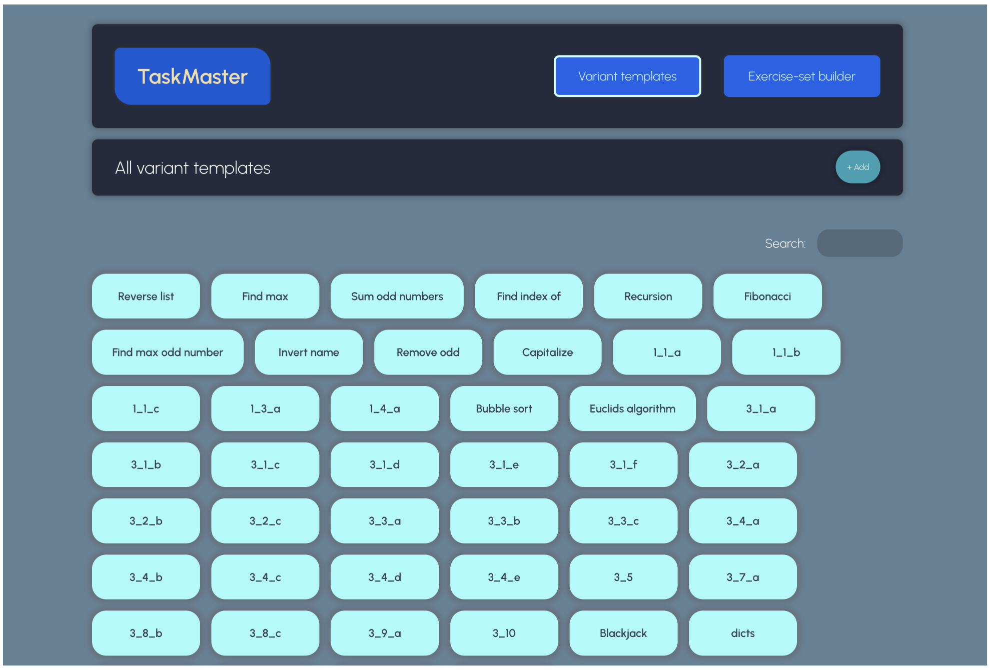


Figure 5.3: The page listing all variant templates

described in Section 2.4.

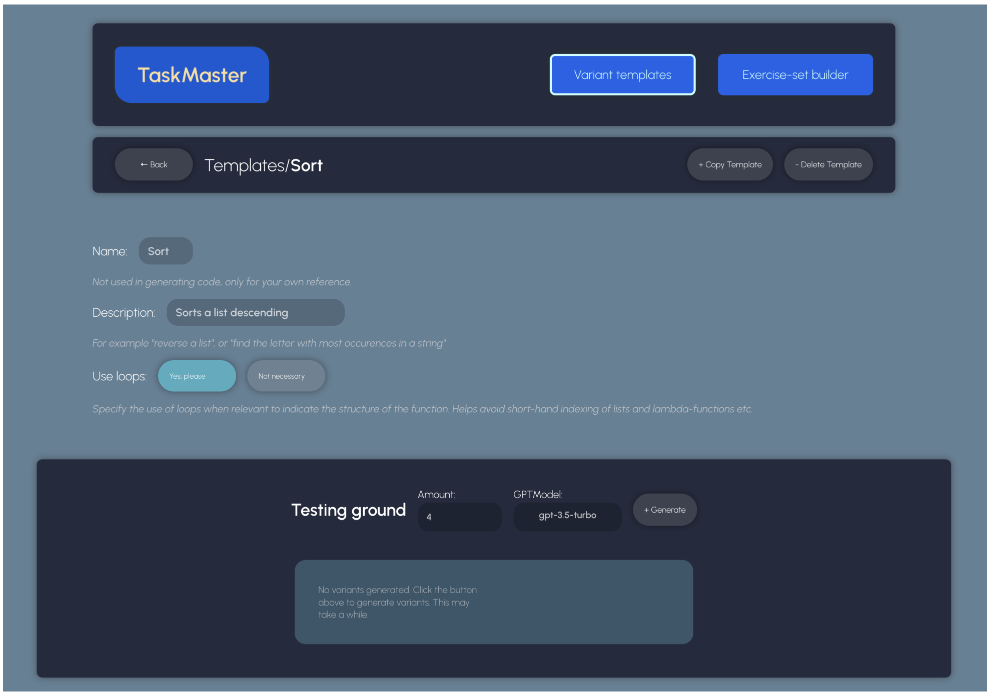
Depending on the requested number of variants, generating a batch of variants may take from 20 seconds to a few minutes. In cases where an unchanged variant template has been used to generate variants previously, caching makes the process instant. To give the user an indication of ongoing generation processes, a "loading box" is displayed for each, as seen in Figure 5.6.

Variant testing ground

As soon as a batch of requested variants is done generating, the resulting variants are listed in the "Testing ground", as shown in Figures 5.7 and 5.8. On the top of the view is a list of meta-properties from the generation process, alongside the standard deviation of each rating metric on the variants. Then follows a vertical list of all the variants, each including the task description, the code, the ratings and an example of execution.

5.2.4 Exercise-set builder for combining the variants

The second sub-page of TaskMaster is the "Exercise set builder", where the user is able to combine any of the variant templates to produce an exercise set. This view is seen in Figure 5.9. At this stage, the relevant variant templates should all be completed and tested in the "Variant templates" view, ready for use in an exercise. For each desired task in an exercise set, the user selects one variant tem-



Name: Sort

Not used in generating code, only for your own reference.

Description: Sorts a list descending

For example "reverse a list", or "find the letter with most occurences in a string"

Use loops:

Specify the use of loops when relevant to indicate the structure of the function. Helps avoid short-hand indexing of lists and lambda-functions etc.

Testing ground

Amount: 4

GPTModel: gpt-3.5-turbo

No variants generated. Click the button above to generate variants. This may take a while.

Figure 5.4: Variant template editor with a testing-ground for building variants

plate and one task format. The list of task formats contains the formats defined in Section 2.2.1. Selected task specifications are added to the list of tasks, seen in Figure 5.10. For each task added to the list, the user is able to specify the order and how many points are awarded for successfully completing it. The user selects the desired amount of exercise sets, and upon clicking the "Generate" button, the backend starts generating variants from each variant template and assembles them into exercise sets. Usually GPT-prompts in the variant generation are already cached from the "Testing ground" earlier in the process, reducing the time-cost of generating exercise sets.

List of generated exercise-sets

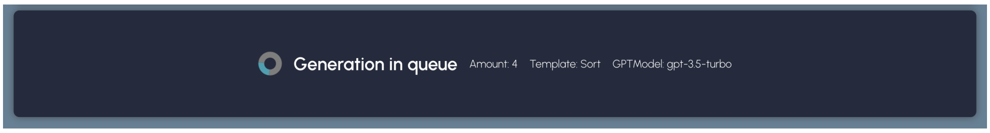
As soon as the exercise sets are completed, they are listed together with the corresponding solution sets, as links to PDF files. This can be seen in Figure 5.11. The PDF files are accessed in the browser by the user, and can be downloaded and stored. From here, the user can click "Back", and modify the list of tasks to make smaller changes and generate again.

5.3 Assessment

This section includes all aspects of user assessment of the system - both in terms of the ability of teaching staff to effectively produce exercise sets, and in turn by

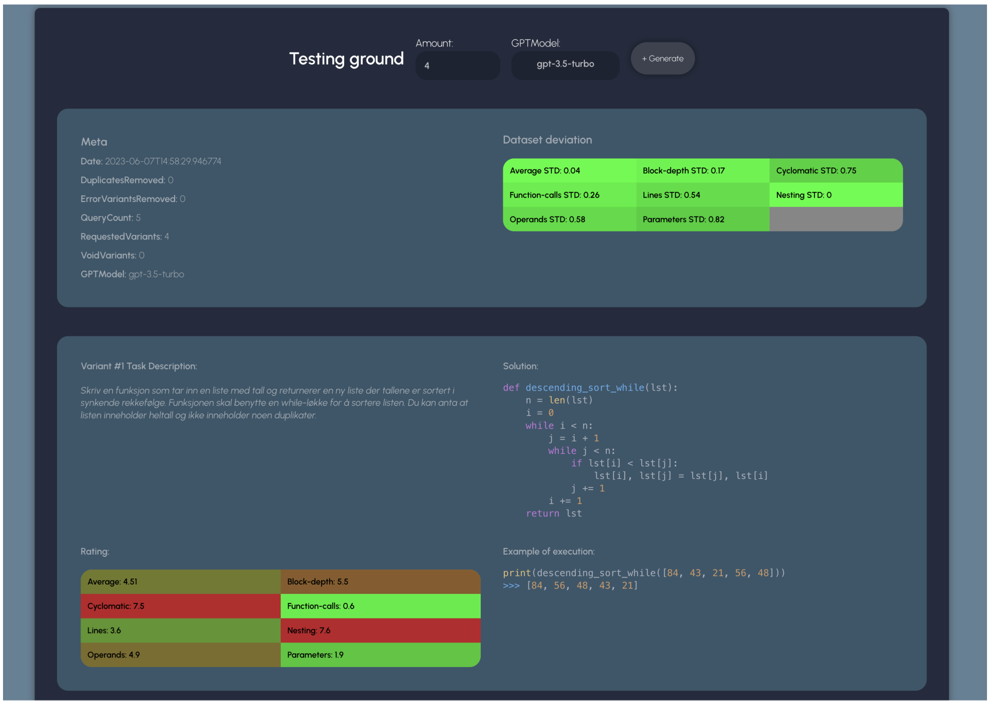
GPTModel: gpt-3.5-turbo

Figure 5.5: Dropdown menu with available GPT-models for generating variants



Generation in queue Amount: 4 Template: Sort GPTModel: gpt-3.5-turbo

Figure 5.6: Indication that generation is in progress



Amount: 4

GPTModel: gpt-3.5-turbo

Meta

Date: 2023-06-07T14:58:29.946774
DuplicatesRemoved: 0
ErrorVariantsRemoved: 0
QueryCount: 5
RequestedVariants: 4
VoidVariants: 0
GPTModel: gpt-3.5-turbo

Dataset deviation

Average STD: 0.04	Block-depth STD: 0.17	Cyclomatic STD: 0.75
Function-calls STD: 0.26	Lines STD: 0.54	Nesting STD: 0
Operands STD: 0.58	Parameters STD: 0.82	

Variant #1 Task Description:

Skriv en funksjon som tar inn en liste med tall og returnerer en ny liste der tallene er sortert i synkende rekkefølge. Funksjonen skal benytte en while-løkke for å sortere listen. Du kan anta at listen inneholder heltall og ikke inneholder noen duplikater.

Solution:

```python
def descending_sort_while(lst):
    n = len(lst)
    i = 0
    while i < n:
        j = i + 1
        while j < n:
            if lst[i] < lst[j]:
                lst[i], lst[j] = lst[j], lst[i]
            j += 1
        i += 1
    return lst
```

Rating:

Average: 4.51	Block-depth: 5.5
Cyclomatic: 7.5	Function-calls: 0.6
Lines: 3.6	Nesting: 7.6
Operands: 4.9	Parameters: 1.9

Example of execution:

```
print(descending_sort_while([84, 43, 21, 56, 48]))
>>> [84, 56, 48, 43, 21]
```

Figure 5.7: Results from generating a batch of variants in the testing-ground


Variant #2 Task Description:

Skriv en funksjon som tar inn en liste med tall og returnerer en ny liste der tallene er sortert i synkende rekkefølge. Funksjonen skal benytte en for-løkke for å sortere listen. Du kan anta at listen inneholder heltall og ikke inneholder noen duplikater.

Solution:

```python
def descending_sort_for(lst):
    n = len(lst)
    for i in range(n):
        for j in range(i + 1, n):
            if lst[i] < lst[j]:
                lst[i], lst[j] = lst[j], lst[i]
    return lst
```

Rating:

Average: 4.43	Block-depth: 5.1
Cyclomatic: 7.5	Function-calls: 1.2
Lines: 2.3	Nesting: 7.6
Operands: 3.5	Parameters: 3.8

Example of execution:

```
print(descending_sort_for([54, 12, 37, 44, 28]))
>>> [54, 44, 37, 28, 12]
```

Variant #3 Task Description:

Skriv en funksjon som tar inn en liste med tall og returnerer en ny liste der tallene er sortert i synkende rekkefølge. Funksjonen skal benytte boblesorteringsalgoritmen for å sortere listen. Du kan anta at listen inneholder heltall og ikke inneholder noen duplikater.

Solution:

```python
def descending_sort_bubble_sort(lst):
    n = len(lst)
    for i in range(n):
        for j in range(n - i - 1):
            if lst[j] < lst[j + 1]:
                lst[j], lst[j + 1] = lst[j + 1], lst[j]
    return lst
```

Rating:

Average: 4.41	Block-depth: 5.1
Cyclomatic: 6	Function-calls: 1.2
Lines: 2.3	Nesting: 7.6
Operands: 4.9	Parameters: 3.8

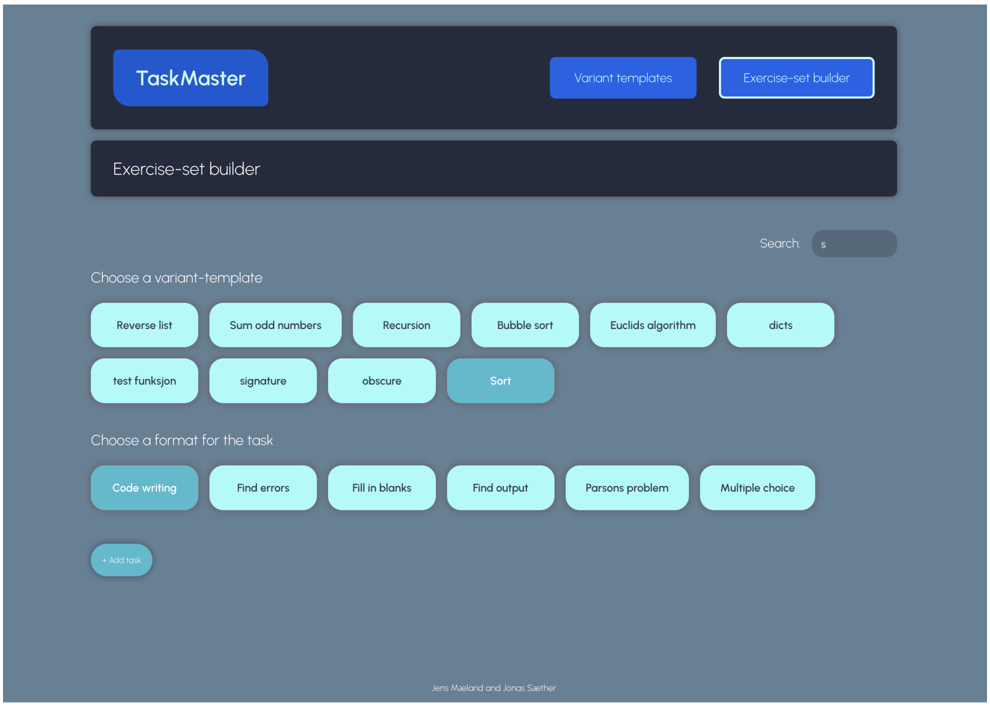
Example of execution:

```
print(descending_sort_bubble_sort([99, 93, 82, 65, 0]))
>>> [99, 93, 82, 65, 0]
```

Variant #4 Task Description:

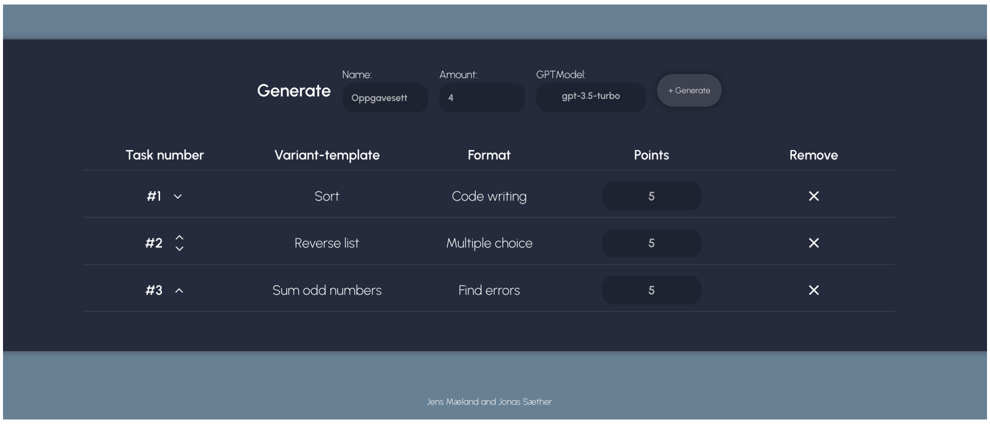
Solution:

Figure 5.8: More results from generating a batch of variants in the testing-ground



Search: s

Figure 5.9: The exercise-set builder, where variant templates are combined with task formats



Name: Oppgavesett
Amount: 4
GPTModel: gpt-3.5-turbo

Task number	Variant-template	Format	Points	Remove
#1	Sort	Code writing	5	×
#2	Reverse list	Multiple choice	5	×
#3	Sum odd numbers	Find errors	5	×

Figure 5.10: The list of selected exercises to be generated in the exercise-set builder

Complete!

Exercise number	Exercise-set	Solution	Average complexity
#1	Oppgavesett_exercise_1.pdf	Oppgavesett_solution_1.pdf	3.41
#2	Oppgavesett_exercise_2.pdf	Oppgavesett_solution_2.pdf	3.04
#3	Oppgavesett_exercise_3.pdf	Oppgavesett_solution_3.pdf	2.49
#4	Oppgavesett_exercise_4.pdf	Oppgavesett_solution_4.pdf	3.03

Jens Mæland and Jonas Sæther

Figure 5.11: A list of complete exercise-sets on PDF, accessible to the user

evaluating the generated exercise sets on students. The assessment tackles the inquiry in Research Question 3[1.2], namely in which settings the generated exercises perform the best.

5.3.1 Student demo

The first assessment in the project is not directly related to using the artefact, but instead assessing the output that it produces. The rationale being that it is necessary to establish the quality and usability of the exercise sets generated before assessing the system itself. For the demo, the participants were therefore not introduced to the artefact at all, only given physical printed exercise sets, simulating a conventional exercise class. The goal was to assess three main points. Firstly, whether the generated tasks are coherent and solvable. This point requires the participants to understand the logic of the problem, the wording of the textual description, and the format in which the tasks are presented. Secondly, assessing whether the exercise sets are of similar difficulty and aligned with the automated ratings. Lastly, we looked at the perceived opportunity for each participant to cheat by looking at another.

To ensure the simulation was realistic, it was decided to include an exercise set from the pool of previously used sets in the introductory course ITGK at NTNU. The selected exercise set was exercise number three out of ten from that year, and contained the segments "Introduction to loops", "More on loops", "Intro to nested loops", "Code understanding", "Guess the number", "Geometric series", "Fibonacci", and "Hangman". The exercise set was selected mainly because of its level of difficulty. It resides on the less difficult part of the ITGK-course material, but is not amongst the most introductory and trivial of exercise sets. It contains a lower degree of contextual case tasks than later exercise sets, yet some references to widely known concepts such as Fibonacci numbers and also more specific ideas such as referencing the course ITGK. Examples of both are seen in Listings 5.9 and 5.10.

Code listing 5.9: Simple example from exercise-set

```
for x in range(5):
    print("Jeg elsker ITGK! Jeg elsker ITGK! Jeg elsker ITGK!")
```

Code listing 5.10: Contextual example from exercise-set

```
The fibonacci-numbers are defined as follows:

f(0)=0
f(1)=1
f(k)=f(k-1)+f(k-2)

That means the first two numbers in the series are 0 and 1, thereafter the next
number is the sum of the two previous numbers. The start of the series therefore
looks like this: 0 1 1 2 3 5 8 13 ...

Create a program that calculates and returns the k-th fibonacci-number f(k) by
using iteration. If successful, the 10th Fibonacci-number should be 34.
```

From the 30 tasks in the exercise set, 24 of them were selected. The remaining few tasks were removed because they were very similar, intended for volume-training, or they were deemed too time-consuming in this specific assessment scenario. We named the selected set of tasks "Demo ITGK". This enabled modelling the demo exercise sets generated by the artefact after the selected existing tasks. Thus variant templates were built for each task in "Demo ITGK". The description in each respective variant template were derived from looking at the tasks, attempting to capture the important elements in a single sentence. The following list contains the description of the 24 tasks selected from the exercise set, created as templates in TaskMaster.

1. "prompt the user for an adjective and range and prints a sentence claiming to be more of the adjective"
2. "prompt the user for an adjective and prints a sentence claiming to be more of the adjective, until the adjective is an empty string"
3. "prompt the user for an adjective and prints a sentence claiming to be more of the adjective, until the total adjective character count is 42"
4. "print all numbers in the 5-multiplication table between 20 and 81, prints numbers from 48 to 80 incrementing by 8, and prints numbers backwards from 100 to 80 incrementing by -3"
5. "print the numbers 1 to 5"
6. "print the numbers 15 to 1 backwards"
7. "prompt the user 7 times for numbers, and prints the sum"
8. "multiply all numbers from 1, ending when the sum is larger than 1000"
9. "prompt the user by asking for the capital of Niue until the correct answer is given, printing the number of tries"
10. "prompt the user for amounts of students and amount of subjects, and prints all combinations of 'Student x loves subject n'"
11. "print all timestamps for one day, on the format HH:MM"

12. "print all multiplication tables between 1 to 10"
13. "floor-divide a variable with the value 345 by 2 until it reaches 0, and adds each value to a string"
14. "print each even number from 0 to 10, and prints "divisible by 4" if that is the case"
15. "multiply a variable with initial value 1 with 2 ten times and prints the resulting value"
16. "multiply a variable with initial value 1 with 2 three times and prints the resulting value"
17. "print a star i + 1 times one one line for x in range i with initial value 5"
18. "prompt the user for a lower and upper limit, and then prompts for a number by telling to go lower or higher until the correct random value is guessed"
19. "sum a geometric series from 0 to n"
20. "find the fibonacci value of a number"
21. "find the sum of fibonacci-numbers up to a provided number"
22. "return a list of all fibonacci-numbers up to the provided number"
23. "find the sum of the series of all numbers squared up to the provided number. Every other number should be subtracted instead of added"
24. "run the game hangman. It starts by prompting the user for a word and number of tries"

Three examples of tasks from the original "Demo ITGK" coupled with a resulting generated variant are shown in Listings 5.11 and 5.12, 5.18 and 5.14, through 5.15 and 5.16.

Code listing 5.11: Task 6 from "Demo ITGK"

```
Lag et program som teller nedover fra 15
til 1 ved hjelp av en for-løkke.

Skriv ut alle tallene.
```

Code listing 5.12: Task 6 from "Demo 1"

```
Skriv en funksjon som tar inn et heltall
som argument (standardverdi 15) og
skriver ut alle heltall fra dette tallet
og ned til 1 i synkende rekkefølge.
Implementer funksjonen ved hjelp av
rekursjon.

Eksempel på kjøring:
print_numbers_recursion(15)
>>> 15
14
13
12
11
10
..
4
3
2
1
```

Code listing 5.13: Task 7 from "Demo ITGK"

```
Lag et program som ved hjelp av en løkke
ber brukeren om å taste inn 7 heltall,
som vist i eksemplet på kjøring under.
Til slutt skal programmet skrive ut hva
summen av tallene ble.

'''
Skriv inn et heltall: 6
Skriv inn et heltall: 4
Skriv inn et heltall: 7
Skriv inn et heltall: 3
Skriv inn et heltall: 2
Skriv inn et heltall: 456
Skriv inn et heltall: 99
Summen av tallene ble 577
'''
```

Code listing 5.14: Task 7 from "Demo 1"

```
Skriv et program som tar inn 7 heltall
fra brukeren, legger sammen alle tallene
og skriver ut summen. Bruk liste-forståelse
for å samle inn tallene fra brukeren.
```

Code listing 5.15: Task 16 from "Demo ITGK"

```
Hva skrives ut i koden under?

i=1
j=3
while j>0:
    i = i*2
    j=j-1
print(i)
```

Code listing 5.16: Task 16 from "Demo 1"

```
Hva er output av følgende kjøring?

def multiply_with_while_loop():
    value = 1
    count = 0
    while count < 3:
        value *= 2
        count += 1
    print(value)

multiply_with_while_loop()
```

To run the demo, 7 participants were selected. These were all current students at NTNU, attending different fields of study. The most frequent study programs were "Cybernetics and Robotics" and "Physics and Mathematics". These programs were relevant because the syllabi includes ITGK, although most of the courses concerns vastly different topics. The goal was to pick participants who are known with the concepts of introductory subjects such as those in ITGK, but not experienced enough to solve all exercises immediately. "Cybernetics and Robotics" is more hardware-specific than for instance the "Computer Science"-study program, while "Physics and Mathematics" is more focused on applying the programming on calculations and numerics.

The participants were selected to cover all 5 years of their respective master programs. Participants on the first year had therefore recently attended the ITGK course, while one participant attended the course 4 years ago. We refer to the 7 participants as "Participant # (Number)". In stead of generating unique exercise sets for each participant, 4 different exercise sets were built. The purpose of this approach was to have more than one participant assess each exercise set, to mitigate subjective feedback and outliers. The four generated exercise sets are referred to as "Demo # (Number)", and distributed amongst 6 of the participants. Participant 7 was given the original "Demo ITGK".

Participant	Exercise-set
Participant 1	Demo 1
Participant 2	Demo 2
Participant 3	Demo 2
Participant 4	Demo 3
Participant 5	Demo 3
Participant 6	Demo 4
Participant 7	Demo ITGK

Table 5.2: Distribution of demo exercise-sets and participants.

The assessment process was structured corresponding to the tasks. In order to conform with the time slot allocated for the demo, 8 of the 24 tasks in each exercise set were selected, namely Task 1, Task 4, Task 6, Task 7, Task 10, Task 14, Task 16 and Task 22. For each task in the exercise sets, the participants answered four questions:

1. Does the task make sense? Was there something unusual about the task or the task description?
2. Did you manage to complete the task?
3. On a scale from 1-10, where 1 is trivial and 10 represents the most difficult one can expect in ITGK, how difficult was the task?
4. Do you miss any information in the task?

Task	ITGK	1	2	2	3	3	4	Average	STD
1	3	3	3	3	3	5	1	3	1,15
4	3	4	4	4	10	4	4	4,71	2,36
6	2	7	2	5	2	2	1	3	2,16
7	4,5	4	3	1	3	3	2	2,93	1,25
10	5	5,5	6	6	7	2	3	4,93	1,64
14	4	5	6	4	4	3	5	4,43	0,98
16	5	5	9	4	6	4	8	5,86	1,95
22	8	7	10	7	9	5	10	8	1,83
Average	4,31	5,06	5,38	4,25	5,5	3,5	4,25		

Table 5.3: Q3: Assessed difficulty of each task in each exercise set.

The key question in the task assessment was Question 3. Responses to the question enables analyzing fairness and equality in difficulty between variants from each exercise set. While the data set is too small to draw any conclusions, Table 5.3 shows that the different variants of each task in many cases show a similar perceived difficulty. There are some outliers in the data set, and as seen in 5.5. A common trait for these outliers are that the participant were not able to

Task	ITGK	1	2	3	4	Average
1	x	3,19	2,71	4	3,16	3,27
4	x	4,51	4,31	5,01	5,06	4,72
6	x	2,49	2,49	2,34	2,61	2,48
7	x	2,57	3,27	3,61	3,31	3,19
10	x	4,26	4,33	4,29	3,69	4,14
14	x	3,59	4,2	3,54	4,53	3,97
16	x	2,7	2,66	2,64	2,03	2,51
22	x	2,49	4,09	2,49	3,51	3,15
Average	x	3,23	3,51	3,49	3,49	

Table 5.4: TaskMaster automated rating of task difficulties in each exercise set.

Task	ITGK	1	2	2	3	3	4
1	Yes	Yes	Yes	Yes	Yes	No	Yes
4	Partially	Yes	Yes	Yes	No	Yes	Yes
6	Yes	No	Yes	Yes	Yes	Yes	Yes
7	Yes	Yes	Yes	Yes	Yes	Yes	Yes
10	Partially	Yes	Yes	Yes	No	Yes	Yes
14	Yes	Yes	Yes	Yes	Yes	Yes	Yes
16	Yes	Yes	Yes	Yes	No	Yes	Yes
22	Yes	Yes	No	Yes	No	Yes	No

Table 5.5: Q2: Participants succeeding at each task in each exercise set.

complete the task, and therefore understandably gave it a high difficulty rating. The results from this question also enables comparing perceived difficulty with the automated rating of each variant by the artefact, consequently assessing the rating system itself. After generating the exercise sets, the ratings produced by TaskMaster were recorded, listed in Table 5.4. While these distinguish between different metrics such as cyclomatic complexity and nesting, the average rating for each task is used when comparing with perceived difficulty by the participants.

Reviewing the "Average" column from both Table 5.3 and Table 5.4, we observe that the first 6 tasks are closely aligned. However, the two most difficult tasks, Task 16 and Task 22, are not correlated with the average ratings produced by the artefact, clearly showing that the automated rating is not accurate for certain tasks. However, the results indicate that there is some correlation between the estimated and observed difficulty. Also, the weights to each rating function, defined in Table 5.1, appears to be quite accurate except for the outliers.

The tasks where the perceived difficulty deviates largely from the average rating produced by TaskMaster often fits into one of two categories: Abstract and obscure code with no context, or very specific concepts that requires knowledge outside of the programming itself. These are both elements that adds to the complexity of the task, but are not considered by the rating methods. An example of obscure code is Task 16 [5.16] from the student demo. Tracing exercises similar to Task 16 are often obscure by design, to increase the difficulty. Task 22 [5.17] from the student demo, on the other hand, is very specific by requesting a list of Fibonacci numbers. This requires knowledge of the Fibonacci series and successfully implementing it in code. For a better automatic rating of difficulty, a measurement of the semantic meaning of the code related to a task needs to be developed.

Code listing 5.17: Task 22 from "Demo 1", showcasing a concept-specific variant

```
Skriv en funksjon som tar inn et heltall ’n’ og returnerer en liste med alle
Fibonacci-tallene som er mindre eller lik ’n’. Fibonacci-tallene er en sekvens
av tall der hvert tall er summen av de to foregående tallene, med de to første
tallene i sekvensen som 0 og 1.

Eksempel på kjøring:
print(fibonacci_loop(33))
>>> [0, 1, 1, 2, 3, 5, 8, 13, 21]
```

After all participants had filled out their individual assessment forms, the variants were assessed in plenary by the group. For each task, the group answered the following questions:

1. The participants read the task descriptions from the other exercise sets. Which task does the group prefer?
2. Which task is the least preferred by the group?
3. By looking at the tasks and solutions of other participants, would the participants be able to cheat on their own task?

It is important to note that "Demo ITGK", the original exercise set that the demo was based upon, was intended to solve using an IDE. Some of the tasks therefore contains code snippets as a basis for the task, where students are supposed to continue and finish the code. This task format is described in Section 2.2.2, and not supported by TaskMaster. Handing out physical copies of "Demo ITGK" therefore breaches with the intentions of the authors. Nevertheless, it enables comparing the task descriptions and formats to the generated exercise sets.

The group preferred tasks from all the different exercise sets, with "Demo 2" being the most desirable. It is worth noting that the preference of a task was interpreted as identifying the "easiest" task according to several of the participant. One clear observation was that students preferred tasks where clear instructions were given, as long as they instructions did not impose the use of difficult concepts. An example of this is Task 7 from the exercise sets, illustrated in Listings 5.18, 5.19 and 5.20, where the task description of the two different variants states different requirements. The group preferred "Demo 2" in this case, and the least desirable was "Demo 3".

Code listing 5.18: Task 7 from "Demo ITGK"

```
Lag et program som ved hjelp av en løkke ber brukeren om å taste inn 7 heltall, som
vist i eksemplet på kjøring under. Til slutt skal programmet skrive ut hva summen av
tallene ble.

‘‘‘
Skriv inn et heltall: 6
Skriv inn et heltall: 4
Skriv inn et heltall: 7
Skriv inn et heltall: 3
Skriv inn et heltall: 2
Skriv inn et heltall: 456
Skriv inn et heltall: 99
Summen av tallene ble 577
‘‘‘
```

Code listing 5.19: Task 7 from "Demo 2"

```
Skriv en funksjon som tar inn 7 heltall fra brukeren ved hjelp av en for-løkke, og
deretter beregner og skriver ut summen av disse tallene.
```

Code listing 5.20: Task 7 from "Demo 3"

```
Skriv et program som ber brukeren om å taste inn 7 heltall. Programmet skal
deretter beregne og skrive ut summen av disse tallene. Du kan benytte deg av
funksjonene ‘input()‘, ‘int()‘, ‘sum()‘ og ‘map()‘ for å løse oppgaven.
```

Perceived possibility to cheat on tasks

For 5 of the 8 tasks, the group concluded that they would either not be able to, or not be confident in cheating by copying solutions from other exercise sets. Table 5.6 lists the results for all tasks in the demo. The answers stem from a majority vote, which in all cases was decided with at least 6 of the 7 total participants in agreement. These findings are interesting, as many of the solutions were in fact equivalent. However, due to small differences in the wording and hints given in task descriptions, the variant tasks successfully mitigates the perceived ability of students to copy each other.

1	4	6	7	10	14	16	22
No	Yes	No	Yes	Yes	No	No	No

Table 5.6: Whether the group was confident at cheating on each task.

5.3.2 Research assistant interview

This subsection describes an interview done with a research assistant regarding TaskMaster, showcasing the complete system. The research assistant is referred to as "they" in the following text.

Interview object A

This section will present an interview done with a research assistant in *TDT4110*

Information Technology, Introduction, the most common introductory programming course at NTNU Trondheim. The goal of the interview was to provide insight into the usability of the system we have made. Before conducting the interview, we designed a questionnaire to loosely base the conversation on. As opposed to the student interviews, we wanted this interview to be semi-structured instead of rigidly based on a questionnaire. Our reasoning for this was to allow the interview subjects to be more relaxed, and to invite them to propose changes or talk about their ideas, which could be implemented in our solution.

Interview object A, hereby referred to as "RA1", is currently a 5-th year master student at NTNU, and was a Research Assistant and lecturer in the course *TDT4110 Information Technology, Introduction*, and a Research Assistant in *TDT4100 Object-Oriented Programming* in the period 2022-2023. They has been involved in the creation of course taskwork for both TDT4110 and TDT4100, as well as the creation of exam sets in TDT4100.

Q1: Is the system intuitive, and is the process of generating variant tasks easy to understand?
All in all, RA1 had very little problems navigating the solution and generating his first tasks. Some misunderstandings took place, namely the fact that reloading the web page results in the interruption of task generation and minor UI-related issues clarified by reading the documentation. The tasks generated seemed reasonable, and looked typical for introductory programming courses.

Q2: Generate an exercise set with a couple of variants with different formats. Is there something missing that stops or limits the creation of tasks?
All in all, no limitations existed for RA1 to not be able to generate variants as they pleased. The conversation pivoted towards explaining how the same task concept can be utilized in a variety of ways when generating tasks. The example utilized the "concept" of asking the user to iterate through a list of numbers, and return a list containing only the prime numbers. This concept was then conceived in the form of a parsons problem, a multiple choice problem, a code writing task and a "find the error" problem. RA1 highlighted the variation of tasks that came from the same concept as very useful, and could definitely be used in TDT4110.

Q3: What do you think of the quality of the tasks generated? How would you feel using generated tasks in course exercises?
Using the generated tasks would work very well in RA1s opinion, especially in the course TDT4110. The reasoning behind this is that TDT4110 is a lower-level introductory course, with more of a focus on syntax and programming concepts. For this, the generated tasks were excellent. When discussing TDT4110, RA1 further elaborated on the necessity of tasks being thematically consistent within TDT4110. Discussing how current language models face a significant challenge in generating thematically consistent tasks due to the issue of context overload,

they explained how they had experienced the same problem when using ChatGPT themself when creating exercise sets for TDT4110.

Q4: How would you feel using generated tasks in exams?
RA1 expressed a clear interest in using the generated tasks as a starting point when creating exams. They did underscore the necessity of manually evaluating the quality of these generated tasks, but also emphasized the convenience of not having to create tasks from the ground up, and instead being able to modify already existing tasks. They also pointed out that there exist significant advantages to not manually creating tasks, as they found that they didn't possess the creativity that language models provide.

Q5: How much time do you believe would be saved by using this system instead of manually creating tasks?
He was certain that the solution would save a significant amount of time. Even though they did not want to give an exact measurement, it was obvious that the task generation system would be very useful in his prior work.

Q6: With nothing particular in mind, do you have any comments regarding the system?
He emphasized that they believed the small variations in the tasks would be very valuable in the context of a home exam, but not as impactful in a traditional school exam. The reasoning for this is the fact that they deemed it more likely that students collaborate at home exams, and that collaboration is much more difficult in a traditional exam setting.

The potential for introducing variations into the coursework from one year to the next, was underscored as a substantial influence on reducing the potential for academic dishonesty among students doing their coursework. In the context of TDT4110, a set amount of smaller assignments are expected to be completed at home on a weekly basis, supplemented by two more substantial assignments in the middle and end of the semester that necessitate in-person completion within a traditional examination environment, utilizing Inspera as a digital platform. They especially expressed his conviction that the system would be particularly beneficial in the generation of the larger assignments intended for completion through Inspera.

They also conceived a unique application for the system, which we had not previously considered. As a RA in TDT4110, they bore the responsibility of conducting weekly assignment lectures. These lectures aim at providing more context for that weeks assignment, and provide students with a deeper understanding of the concepts needed to solve it. This is typically done by the RA solving assignments on a blackboard, explaining their reasoning and the process along the way. RA1 believed the system would be perfect for generating this kind of task, and would almost eliminate preparation time needed for the lectures.

Research assistant interview takeaways

After the discussion, we analyzed how the interview had gone, and the main takeaways. The conclusion was simple, the task generation system did what we wanted it to do, and it was intuitive enough to the point that RA1 could use it on his own. RA1 believed the quality of the tasks generated was high enough to the point that they could reliably be used in the exercise sets of courses, especially TDT4110. TDT4100 requires more thematically coherent tasks, which requires more context than current language models are able to handle. Contrary to what we thought, they didn't believe introducing variations would lead to a reduction in cheating done by students completing weekly assignments. His argument was that if they really wanted to cheat on the weekly assignments, small variations in the tasks would not pose a significant problem. After all, they could simply use the same language model used to generate the assignments, to complete the assignments.

As a finishing note, RA1 emphasized that there exists an actual use case for a system like this. RA1 provided valuable insights into the system, and together we created the following list of suggestions to improve the solution:

- Implementing a queue-based system in the API would make it so that refreshing the webpage does not interrupt the generation of variants. The generation of variants would instead be added to a queue, and processed as soon as the API was available.
- Introducing parallelization of task generation would dramatically reduce time spent to generate tasks.
- Changing the design of the "playground" to avoid being misinterpreted to a classic website footer.
- When viewing a template, it would be nice to see some sort of log of what has been generated using that specific template. This would be very nice to gain a bit more understanding regarding the tasks that would be generated.

5.3.3 Professor interviews

This section details the interviews undertaken with professors at NTNU, who hold significant roles in instructing students in introductory programming courses. The interviews were conducted in an informal yet targeted manner, with the central objective being to comprehend the professors' perception of the software's appearance, its potential capacities, and potential future developments. These viewpoints provide a contrast to those of research assistants or students. Each interview involved presenting the current iteration of the solution to the professors, followed by discussions elucidating the system's potential utilization. Each of the professors are referred to as "they" in the following text.

Interview object B Interview object B, hereby referred to as "P1", is a distinguished professor engaged extensively in both introductory and advanced programming education, serving as a lecturer in an array of programming-centric courses. The interview was conducted as described above, and the subsequent paragraphs will provide a summary of P1's perspective on the system.

Context and quality

During the discussion concerning tasks generated by our system, the subject of task context arose. The tasks we provided to P1, generated by our system, were devoid of any specific context, and instead posed straightforward programming questions. In contrast, when crafting coursework for programming courses, the tasks are often embedded within a context—be it a novel card game, a workplace challenge, or something similar. This approach supplies students with a practical framework, facilitating a more comprehensive understanding of the situation and the queries posed by the task.

The topic of task quality in programming also surfaced during the discussion, highlighting the differing quality expectations between students and professors. As P1 elucidated, professors often guide students through task solutions, thus the imperative of exemplifying high standards becomes apparent. For instance, if P1 suggested a solution utilizing inadequately named variables and functions, it inadvertently sets a precedent for students, who may consequently adopt poor naming conventions in their own proposed solutions. P1 summarized their thoughts in an elegant sentence: "Code is not written for the machine, it is written for the next person to read the code". P1 advocated a cautious approach towards the use of automatically generated content, underscoring the necessity for rigorous quality assurance.

Expanding on this notion, it can be assumed that the quality of instructional materials, in this case, the programming tasks, greatly influences the learning outcomes. Given the influence of professors' practices on students' approaches to problem-solving, the importance of quality assurance in programming tasks is elevated. It not only ensures that the tasks are effectively challenging for the students, but also guarantees that they perpetuate best practices and standards in programming. This attention to detail could serve as a critical factor in shaping the students' understanding and competency in programming, which in turn, could significantly impact their performance and success in their future endeavors. However, P1 believed the system could definitively be used to not have to create tasks from the ground up, and instead use them as a starting point and modify them to have sufficient quality. The creativity provided by the system was also interesting to see, and was described as useful.

System expandability and adaptability

When discussing how we made the system, the topic of expandability and adaptability came up. As the system is yet to be fully realized, we expressed our interest in obtaining P1's insights concerning potential future trajectories for the software system. P1, was understandably most interested in the applicability of the system from the viewpoint of a professor, and therefore questioned the ease of incorporating new functionality and expanding the system towards becoming a task generation platform dedicated to assisting professors in the creation of coursework. We clarified that, given the modular nature of the system, the conception of such a professor-oriented system is a plausible future direction. P1 further expounded that more flexible and expandable systems would be more likely for future master's students to enhance and extend, in contrast to more integrated systems that offer limited scope for modification and expansion. The discussion highlighted the importance of developing with scalability and adaptability in mind. We agreed that it could help ensure the systems longevity and relevance in the educational landscape.

Academic dishonesty

P1 explained how they worked with their peers to deter and counteract academic dishonesty, specifically cheating in coursework. We explained our research concerning students' perception of the code tasks generated by our system and presented the relevant data we had collected. P1 argued, however, that the efforts to prevent cheating might be to no avail, considering that students will devise ways to cheat regardless of the structure of the coursework if they really wanted to. Based on this reasoning, P1 explained the stance of prevention by fostering an ethical attitude among students instead of relying on detection and punishment of cheaters.

In the context of examinations, however, P1 endorsed the strategy of having a variety of exam sets to select tasks from, thus creating unique exam task sets for each student. P1 agreed that this tactic can serve as a hurdle for cheaters.

P1 interview takeaways To summarize, our discussions with P1 focused on several critical aspects of our system's development and application in an academic setting. The issue of task context in our system's generated tasks was raised, highlighting the need to embed such tasks within a relevant context to facilitate student understanding. The topic of task quality was addressed, where it was obvious that quality assurance is essential if one were to use automatically generated tasks. However, it was acknowledged that the system's potential as a starting point for task creation was useful.

P1's insights provided valuable guidance for the systems potential applications in an academic context, and possible future work.

Interview object B and C Interview object B and C, hereby referred to as "P2" and "P3", are both renowned professors heavily involved in both introductory and advanced programming education, performing lecturing duties across a spectrum of programming-related courses. The interview was conducted the same was as with P1, except the interview were done on both professors simultaneously. This allowed us to listen in on their shared reasoning and discussions regarding our solution and ideation as we demonstrated functionality. The subsequent paragraphs will provide a summary of P2 and P3's perspective on our system.

Similar discussions as with P1
Several perspectives shared by P2 and P3 aligned with viewpoints encountered when interviewing P1. All three professors were very aligned when it came to their viewpoints regarding academic dishonesty policies, system expandability and adaptability, and the importance of task quality assurance. To avoid redundancy, the overlapping viewpoints that contained the same arguments are left out and the discussions regarding future work are emphasized instead.

Compatibility with existing technologies
One thing that was highlighted as we discussed the system, was intercompatibility with existing technologies. This is an interesting topic, as our system currently does not provide all the functionality needed from task creation, distribution to students, and a student-focused development environment, integration with other solutions is a logical next step. As NTNU uses Inspera, an online assessment platform, P2 and P3 both mentioned their desire for the system to be integrated with the assessment platform. We mentioned that we had already considered such an integration but had not yet reached that point. We also mentioned in the same way as to P1 that the "lecturer" viewpoint, while important, represents just one among many potential future directions for the system.

Creativity and regularity
When looking at the tasks produced, a discussion regarding the balance between creativity and regularity arose. The balance between creativity of a task and the regularity of a task is a fine line. Forcing a language model to focus too much on the structure of a task seems to understandable impede its "creativity". On the other hand, allowing for too much creativity could result in tasks that are unstructured and challenging to utilize within a traditionally structured educational system. After explaining our approach of temperature adjustment based on number of tasks requested, P2 and P3 agreed that it was a good way of handling it.

Future work
During the task generation process, the OpenAI API was extremely slow in responding to prompts made from our system. Understandably, this delay presents a substantial inconvenience for users, lowering the overall user-friendliness of our system. Purchasing a paid subscription or similar service, provided it ensures expedited API response times, would be a worthwhile investment for enhancing the usability of the system.

During the task generation process, some tasks generated were close to perfect, with P2 wishing to simply rename a variable. It then became apparent that the capability to directly edit the programming tasks generated within the web browser was not in the system. Both P2 and P3 agreed that this feature would be a valuable addition to the system. The discussion progressed to highlight what they felt was the Inspera platform's inadequate task editing functionality, where they agreed that if our system could supply this feature, it would substantially enhance the overall utility of the program.

To summarize, both P2 and P3 expressed genuine interest in making use of such a system to generate tasks which can be used as starting points for more complex and "human" tasks. Both professors already made use of ChatGPT when creating coursework, and expressed interest in this form of structure added to a language model.

Chapter 6

Discussion

This chapter discusses the process and results gathered during the design science process, with respect to the research questions.

6.1 Summary of the variant task generation system

Through an agile development process, a system for automatically generating programming tasks using GPT-models has been built. The resulting system exhibit many useful features in effectively generating programming tasks. While simpler approaches such as using the large language models directly without a system would also prove useful, the assessment process has shown the increased efficiency by using the software in many use-cases. The combination of query design, automatically structuring and rating the variants, as well as the ability to create exercise sets using a GUI enables a seamless process of task-generation. However, the system has limitations. The variants generated are not always of high quality, and the users have limited ability to manipulate the output. Thus a more flexible structure could prove beneficial in further development.

6.2 Comparisons to prior work

As discussed in chapter 4, there have been related work conducted which studies different ways of generating programming tasks using large language models. This chapter compares our work to related work. There are several differentiating aspects such as language models used, how the language models are queried, and how the system surrounding the actual task creation is handled.

6.2.1 Query technique

The technique used to query the language models differ considerably between our work and related work. As described in figure 4.1, Sarsa et al.[52] uses a different approach to querying than TaskMaster. They make use of a large and thorough description of a task to prime the language model into providing them with a novel second and structurally similar task. The same thorough description can be reused several times, as the goal is for the language model to gather inspiration from the structure of the priming example.

There are a few reasons why this approach is not ideal when implementing a structured system meant to generate tasks on a large scale. When Sarsa et al. primed OpenAI Codex, they provide a multitude of data created by a human being, and use two fairly similar human-made exercises. Although the strict structure of the docstring results in a somewhat standardized output format, the context that must be attached to each task makes it difficult to create differing tasks in large quantities. If TaskMaster were to implement the usage of similar priming, large sets of programming tasks of varying complexity and structure, solutions and tests would need to be written manually and be implemented in the priming of the system. There are many different task formats, and this approach would require at least one task showing the detailed structure for each of them in order to work. TaskMaster is also a prototype built upon the wish of *effortlessly* creating programming tasks, which in its current state is reflected by the fact that an enormous amount of programming tasks can be generated from a simple description.

There are also clear advantages to using similar priming. As described in chapter 5.3.3, programming tasks handed out to students are typically set within a context meant to aid students in understanding the goal of their task. Being told to write a function describing a familiar action (i.e. purchasing a bus ticket with a given discount) is preferred when compared to plain, generic functions. For TaskMaster to support the functionality needed to make similar docstring priming viable for large amounts of coding tasks would take more time than we had at our disposal.

6.2.2 Task quality and readiness of use

The tasks generated by TaskMaster face similar challenges and characteristics as generated programming tasks found in similar work. Although we did not conduct a formal analysis, the level of "readiness of use" observed in our generated tasks aligns with the findings reported by Sarsa et al.[52]. Feedback received from professors regarding the generated tasks was very clear: tasks must be more refined before being presented to students. However, there was not a lot of work required before tasks could be used, and there was consensus that the generated tasks served as excellent starting points when developing coursework.

6.3 Limitations and threats to validity

The purpose of this chapter is to explore and articulate the limitations of our framework. The following sections provides discussions on some of the identified limitations of our work.

6.3.1 LLMs and large amounts of context

While the current generation of LLMs demonstrates remarkable capabilities in generating code snippets and performing simple programming tasks, their capacity to construct complex, context-rich programming tasks are not fully matured. Embedding a function or task within a broader, more realistic concept requires a level of contextual understanding and knowledge of the real world that LLMs are currently limited in. It is important to know that LLMs lack a genuine understanding of real-world entities, concepts and relationships. They don't possess intrinsic knowledge about the world, they rely on patterns they've learned from the data they have been trained on.

LLMs and user intent clarification

Current LLMs lack the capability to interactively clarify the requirements or intent behind a prompt with an end user. As described in section 2, they operate on a feed-forward mechanism, producing an output based on an input without the possibility for back-and-forth clarification. Due to this, the intent of an end user might be "lost in translation" between input and output.

6.3.2 Is the system redundant?

As previously highlighted in section 5.3.3, the professors we interviewed have previously used ChatGPT to aid their manual process of generating programming tasks. Even though our system contributes with much more than just asking ChatGPT to create tasks, there are some arguments to be made regarding if the system is redundant or not.

The tasks created by our system often need further refinement before they can be applied directly in a teaching environment, and definitely needs manual quality control. This necessity for modification for each task generated makes us wonder if some faculty would be bette off manually using ChatGPT instead, if what they seek is the creativeness of the LLM and don't care for the functionality regarding exercise set generation or variant generation.

6.3.3 The use of both agile and waterfall method

During the developmental phase of our system, we initially perceived our approach to be aligned with an agile framework, specifically an adaptation of Scrum. However, upon reflection, it is apparent that our methodology mirrored a hybrid of the Scrum and waterfall methods. We used the key values of scrum, but the progression and structure of our three development sprints leaned towards the waterfall method.

Each of these sprints focused on a distinct part of the same system. The first sprint concerned the integration of the OpenAI API and the establishment of the variant task generation system, while the second sprint was centered around the creation of a rating system and an exercise builder. The third sprint was aimed at the development of a REST API and frontend solution.

This division of work and the completion of each subsystem before moving onto the next mirrors the linear nature of the waterfall method. We believe this happened due to not having a fully concrete vision of the final system from the onset. Given the small size of our team, which comprised only two individuals, we deemed it more efficient to entirely finish one subsystem before proceeding to the next. This decision was primarily driven by a desire to avoid extra work that might come from needing to implement changes in already completed components to accommodate the later stages, something that was actually mostly avoided.

To summarize, our approach were a balance somewhat making use of the iterative flexibility of Scrum and the sequential progression inherent in the waterfall method. We still believe the right choice was to work this way, as we believe it has led the system further than what would have been possible with pure scrum. It is also an insight into the practical realities of system development.

6.3.4 Lack of direction

Due to the fact that TaskMaster is a project made by exploring the capabilities of generating programming tasks using large language models, the system is not angled towards a specific userbase. We received feedback to further confirm this during the end user interviews conducted. An important drawback stemming from this approach is the limited visibility into the system's complete capabilities from the perspective of, for instance, a professor. On the other hand, the absence of a specific user focus has resulted in valuable data from multiple end user groups, and the development of a modular system that can adapt to diverse future requirements. The system was also designed with a focus on modifiability, deliberately avoiding design choices that would hinder certain future development directions.

6.3.5 Subjectivity of complexity

An underlying problem in a system which generates metrics based on a programming task, is the subjectivity of complexity. The perceived complexity of a programming task is subjective, and depends fully on a programmers experience, skill level, and most importantly familiarity with the programming language or problem domain. Even though the system assigns a complexity score based on objective metrics, these don't necessarily align with the perceived difficulty for individual end users. The fact that the systems displayed values are weighted based on an introductory programming course, will also enforce a bias into the metrics system only making it viable for introductory programming courses. However, the underlying metric functions can be used to create weighted systems more fitting for more complex programming courses.

6.3.6 Multiple equivalent solutions in relation to complexity rating

A common occurrence in programming tasks is that there are multiple equivalent solutions to a programming task. A naive metrics system, in the sense that it is based on line-by-line processing, will give assign a higher difficulty rating on an unnecessarily line-heavy solution compared to a sleek solution using list comprehension. An example of this can be seen in the code snippets below, where the functionality is equivalent but the road to get there is different.

```
numbers = [1, 2, 3, 4, 5]
squared_numbers = []
for number in numbers:
    squared = number ** 2
    squared_numbers.append(squared)
```

```
numbers = [1, 2, 3, 4, 5]
squared_numbers = [number ** 2 for number in numbers]
```

However, we still chose to not create a more complex metrics system. More sophisticated metrics would require more advanced algorithms and statistic techniques to implement. This would lead to a significant time investment in both development and testing, which was better used on the generation system. Another argument is the fact that the system is designed for use in introductory CS courses, which makes it plausible that users of the system are of a similar, relatively low skill level. It was deemed likely that the number of times the system encountered such discrepancies as in the example above would be relatively small, and therefore does not need a more complex metrics system.

6.3.7 Weighted complexity measure

As outlined in section 3.3, our metrics are derived from a weighted complexity analysis of the programming task presented to the user.These weights serve the purpose of standardizing raw complexity measures, such as ’number of lines,’ into a more universally understood scale of 1 to 10. While the goal of this standardizing process was to fit our difficulty spectrum into classic CS1 tasks, biases stemming from the specific tasks process has made its way into these weights. The weights therefore not only reflect the complexity of the tasks, but also the specific characteristics of the tasks that were used to calibrate the system. There are also many different ways of squeezing non-negative into an interval between 0 and 10. We selected the approach which we found most intuitive, but there might exist better ways of doing it.

6.3.8 Biased measurements due to code base size

The size of a solution, as measured by lines of code, is one of the most basic measurements one can make about a piece of software. A problem with the rating system as it is now, is the fact that most of the metrics tracked significantly correlate with the size of the code snippet in question. If a solution has many lines of code but these lines are mostly repetitive or straightforward, our metrics system might overvalue its complexity compared to what a human being might think. Similarly, small solutions which make use of "complex" language functionality such as lambda expressions or built-in modules may be unjustly measured to be less complex than the equivalent code snippet written out using verbose code.

6.3.9 A new landscape

Navigating the landscape of systems akin to our presents certain challenges, as such technologies have only become feasible in the recent past. As the development and utilization of frameworks that incorporate LLMs continue to evolve, effective strategies to manage AI are still being refined. This emphasizes the potential for this technology to grow significantly more complex in the future, something that should be kept in mind when reviewing past works within this rapidly evolving field.

6.3.10 Small sample size and realism of testing

The task generation system underwent testing with a small sample of seven individuals, who willingly participated in a mock examination without any compensation. These participants, recruited from acquaintances and mutual connections, completed exercise sets composed of 50 variant templates.

Such a sample, while convenient, may introduce several biases. The selection bias is a concern as the participants, all students at NTNU, may not represent the larger relevant population, which would be introductory CS students. Their relationship to us, even if indirect, may also introduce a familiarity bias. Despite these potential biases, we are confident in the honesty of the feedback received, as the participants demonstrated their willingness to provide critical responses.

We also did not get to examine how actual students in an ongoing introductory computer science course would handle the tasks generated. This omission was due to a combination of timing, with the project's completion coinciding with the end of the semester, and the unavailability of any suitable programming courses during that period.

It is also important to note the fact that no testing was done in a real-world learning environment. We did therefore not get to see how actual students in an actual introductory computer science course would handle the tasks generated. The reason for this was both that no introductory programming courses were running that semester, and the solution was completed at the end of the semester.

However, due to these conditions, especially the limited sample size, no substantive statistical analysis could be conducted on the examination data, hence the lack of statistical interpretation in the Results section. This context should be kept in mind by readers when considering the implications of the findings presented.

Chapter 7

Conclusion and Future Work

This chapter marks the conclusion of our thesis, bringing together the many threads of our investigation and providing an answer to the research questions. We will also outline possible future research, drawing from the thesis limitations, unanswered questions and new questions which have emerged during our research.

Research Question 1 (RQ1): How can a software system be structured in order to allow effortless generation of programming tasks using OpenAI GPT-based language models to be readily applicable?

Throughout the project, a prototype software system was developed to determine potential ways of designing and structuring a system in order to allow effortless generation of readily applicable programming tasks. By utilizing the available python package from OpenAI, and developing a modularized system based on the principles of object-oriented programming, a logical structure has been implemented.

The study conducted showed that while promising, language model technology is not yet ripe for use in the generation of programming tasks without necessitating quality assurance. As outlined in 5.3, manual surveillance is required to ensure task quality before making use of the tasks generated.

In designing a software system where the goal is to allow effortless generation of readily applicable programming tasks, it is crucial that the generation of tasks, textual content, tests and potential additional elements be AI-driven. This approach is supported by related work, that suggests AI is the most effective mechanism for generating tasks that emulate human-like qualities. Consequently, the integration of AI in this context becomes not only advantageous but imperative to achieving the system's goal.

Research Question 2 (RQ2): What are the main challenges of developing a system meant to generate and assess the complexity of programming tasks?

Numerous challenges arose during the development of the system artifact. One important challenge was designing high quality queries for the large language models. The prompts being utilized largely affects the responses from the LLMs, which serves as a basis for all variant tasks generated. Building the user experience for task authors also proved a challenge, because of the intricate balance between flexibility and simplicity. Less input by the user meant a simpler process, while more input yielded more control of the task generation. Each design decision in the generation system dictated the features available to the end user.

As for the complexity assessment system, the main challenges consists of putting together a sensible collective metric for complexity, and developing a system capable of measuring said metric. The system implemented is rooted in related work, and does a good job at estimating the complexity of a programming task. However, as outlined in Chapter 6, there are still flaws to the system. Especially the design of complexity measures analyzing task text is interesting, and is outlined in Section 7.2.

Research Question 3 (RQ3): What different use-cases does automated generation of variant tasks have, and in which settings does it perform best?

The automated generation of variant tasks has a variety of different use-cases. Through both literature studies and end user interviews, the use-cases found are the following:

- Generating coursework to be used by course staff in periodically handed out assignments.
- Generating tasks and exercise sets to be used by course staff in mid-semester quizzes.
- Generating tasks and exercise sets to be used by course staff in an exam setting.
- Generating tasks and exercise sets that could be used by students as volume training to improve their understanding of basic concepts.
- Generating tasks that can be used by instructors as inspiration when manually creating coursework.

As for the setting in which the system performs best, further evaluation of the artifact is necessary. While demonstrations on teaching staff with different roles and backgrounds were conducted, the student demonstration and evaluation was more limited. Regrettably only one scenario was examined, namely handing out assignments in the shape of exercise sets. The evaluation provided valuable data on the quality of generated tasks, but is lacking in terms of comparing use cases. Potentially valuable data could have been gathered by focusing on evaluating generated tasks in mid-semester and exam settings, as well as for volume training during courses. However, this was not feasible for us due to our schedule for system finalization.

Automated generation of variant tasks could also be of use in other systems, provided an integration between the systems existed. An integration towards Inspera Assessment was postulated by interviewees as especially interesting, and has been detailed further in 7.2.6.

7.1 Summary of main findings

Our work supports the earlier findings demonstrating that large language models perform well, even in cases where they have not previously encountered similar tasks, or have encountered similar tasks only a handful of times. Our work suggests that current LLMs provide a vast array of opportunities for course staff, and can provide considerable aid when creating, explaining and solving programming tasks. Although there are still a multitude of challenges as discussed in both chapter 4 and 6, we have shown that generating multiple novel and human-like programming tasks is possible by building a system powered by OpenAI language models. We also expect the capabilities of generative AI to only improve over time, allowing for more complex systems to be built around them.

7.2 Future work

This section outline the potential directions for further research and development regarding our study.

7.2.1 Universal Design

Even though the system was developed with a focus on inclusitivity, steps must be taken to ensure that the system is in line with universal design guidelines before it is released. This can be carried out by performing a comprehensive review and usability testing to identify any areas in which the system still falls short of design standards, and improve them.

7.2.2 Query Injection

Given the nature of our system as an exploratory prototype, the emphasis on software security has not been emphasized at all during both the planning and development stages. This potentially exposes our system to a form of security threat, similar to SQL injection, where users could send malicious queries trough our system. The resulting code from such queries could potentially be executed within the system when generating example inputs and outputs for tasks.

Despite OpenAI's measures to handle and discard queries of malicious intent, methods to bypass or "jailbreak" their language model's security mechanisms are frequently discovered. The implementation of appropriate isolation techniques, aimed at safely executing the Python code generated, is both possible and necessary. It is highly recommended that these security measures are incorporated before the system is made accessible to end users, thereby ensuring the overall integrity and reliability of the system.

7.2.3 Expanding system to other programming languages

For the sake of usability in programming courses making use of programming languages other than Python, expanding TaskMaster to support other programming languages would drastically increase the user base capable of using the system. As the system is heavily modularized, not much development is needed to extend the system to other programming languages. What is needed, is a new code deserializer and a rating system, as they are the only components that are built specifically with Python syntax in mind.

7.2.4 Improved solutions using LLMs

While the generation of solutions associated with a task is automated, there is still room for improving the quality of the solutions. In the current state, the artifact only includes an example of the correct code as solutions to the programming tasks. However, there is potential to increase the application of LLMs by incorporating textual explanations of the code. This additional feature could help students understand the proposed solutions, fostering their comprehension and contributing to a richer learning experience.

7.2.5 Utilizing other complexity measures

As seen in 3.3, the complexity score given to a task is based on the analyzed complexity of the proposed solution to a given task. Surveying ways of analyzing the complexity of textual task text, such as number of concepts included in a task, could provide a more accurate difficulty assessment than currently available.

7.2.6 System expansion

System expansion is a natural next step in the process. Feedback from end users indicate that the features providing the most immediate value would be the automated generation of tests accompanying the programming tasks. Both prior research and our empirical research shows the feasibility of utilizing the same language models to generate tests.

Other expansions requested by end users include an integration with Inspera Assessment using the QTI format, and the possibility for task editing and execution directly in the browser. Alternatively, a proprietary "editor" where task authors are able to modify the code or task description of any generated variant would improve the usability of task generation. In addition, the ability to remove any of the generated variants, or to supplement with manually created variants further increase flexibility. To our knowledge, there are no significant obstacles hindering the implementation of these features.

7.2.7 Final thoughts

The focus of this report is primarily concentrated on the design and development of the variant task generation system. However, examining the performance and perception of automatically generated tasks within a genuine academic environment would present intriguing insights.

The implications of systems similar to our extend beyond the scope of this study. The educational landscape is rapidly changing, with AI technology becoming more entrenched in our daily activities. Future systems of a similar nature will also be capable of utilizing AI on an even larger scale, as AI continues to evolve and become more sophisticated. We believe systems as the ones described one day will form a pivotal part of educational infrastructures.

The marriage of AI and education has the potential to foster individualized learning, ensuring each student can learn at their own pace and level of complexity. In this context, systems similar to TaskMaster could play a significant role in the years to come.

Bibliography

[1] D. Radosevic, T. Orehovački and Z. Stapic, 'Automatic on-line generation of student's exercises in teaching programming,' vol. 1, Sep. 2010.

[2] S. Davis, C. Grover, A. Becker and L. Mcgregor, 'Academic dishonesty: Prevalence, determinants, techniques, and punishments,' *Teaching of Psychology - TEACH PSYCHOL*, vol. 19, pp. 16–20, Feb. 1992. DOI: `10.1207/s15328023top1901_3`.

[3] B. Ozmen Yagiz and A. Altun, 'Undergraduate students' experiences in programming: Difficulties and obstacles,' *Turkish Online Journal of Qualitative Inquiry*, vol. 5, Mar. 2014. DOI: `10.17569/tojqi.20328`.

[4] OpenAI, *Openai chatgpt blog post*, `https://openai.com/blog/chatgpt/`, Accessed: 2023-04-30.

[5] A. Baist and A. Pamungkas, 'Analysis of student difficulties in computer programming,' *VOLT : Jurnal Ilmiah Pendidikan Teknik Elektro*, vol. 2, p. 81, Oct. 2017. DOI: `10.30870/volt.v2i2.2211`.

[6] E. Lahtinen, K. Ala-Mutka and H.-M. Järvinen, 'A study of the difficulties of novice programmers,' vol. 37, Sep. 2005, pp. 14–18. DOI: `10.1145/1067445.1067453`.

[7] J. Mæland and J. Sæther, 'Automated generation of programming variant-tasks,' *TDT4105 - Datateknologi, Fordypningsprosjekt*, p. 31, 2022.

[8] V. Iyer and C. Zilles, 'Pattern census: A characterization of pattern usage in early programming courses,' ser. SIGCSE '21, Virtual Event, USA: Association for Computing Machinery, 2021, pp. 45–51, ISBN: 9781450380621. DOI: `10.1145/3408877.3432442`. [Online]. Available: `https://doi.org/10.1145/3408877.3432442`.

[9] P. Brusilovsky, 'A framework for intelligent knowledge sequencing and task sequencing.,' Jan. 1992, pp. 499–506, ISBN: 978-3-540-55606-0. DOI: `10.1007/3-540-55606-0_59`.

[10] R. Pelánek, T. Effenberger and J. Čechák, 'Complexity and difficulty of items in learning systems,' *International Journal of Artificial Intelligence in Education*, vol. 32, no. 1, pp. 196–232, Mar. 2022, ISSN: 1560-4306. DOI: `10.1007/s40593-021-00252-4`. [Online]. Available: `https://doi.org/10.1007/s40593-021-00252-4`.

[11] P. Liu and Z. Li, 'Task complexity: A review and conceptualization framework.,' *International Journal of Industrial Ergonomics*, vol. 42, pp. 553–568, 2012. DOI: `10.1016/j.ergon.2012.09.001`. [Online]. Available: `https://doi.org/10.1016/j.ergon.2012.09.001`.

[12] N. Kasto and J. Whalley, 'Measuring the difficulty of code comprehension tasks using software metrics,' in *Proceedings of the Fifteenth Australasian Computing Education Conference - Volume 136*, ser. ACE '13, Adelaide, Australia: Australian Computer Society, Inc., 2013, pp. 59–65, ISBN: 9781921770210.

[13] IBM, *Rationale software analyzer*, https://shorturl.at/uCJKP, Home Page. Accessed: 2022-02-10.

[14] L. Floridi and M. Chiriatti, 'Gpt-3: Its nature, scope, limits, and consequences,' *Minds Machines 30*, pp. 681–694, 2020, `https://link.springer.com/article/10.1007/s11023-020-09548-1` (visited: 2023-03-03).

[15] OpenAI, *Openai models overview*, `https://platform.openai.com/docs/models/overview`, Accessed: 2023-05-04.

[16] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser and I. Polosukhin, 'Attention is all you need,' in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan and R. Garnett, Eds., vol. 30, Curran Associates, Inc., 2017. [Online]. Available: `https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf`.

[17] J. Alammar, *The illustrated transformer*, `https://jalammar.github.io/illustrated-transformer/`, Accessed: 2023-05-18.

[18] Google, *Google transformer blog post*, `https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html`, Accessed: 2023-05-22.

[19] GOogle, *Tensor2tensor notebook, visualizing self-attention.* `https://colab.research.google.com/github/tensorflow/tensor2tensor/blob/master/tensor2tensor/notebooks/hello_t2t.ipynb`, Accessed: 2023-05-21.

[20] K. Miura, 'An introduction to maximum likelihood estimation and information geometry,' *Interdisciplinary Information Sciences (IIS)*, vol. 17, Nov. 2011. DOI: `10.4036/iis.2011.155`.

[21] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave and G. Lample, *Llama: Open and efficient foundation language models*, 2023. DOI: `https://doi.org/10.48550/arXiv.2302.13971`. arXiv: `2302.13971 [cs.CL]`.

[22] J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. de Las Casas, L. A. Hendricks, J. Welbl, A. Clark, T. Hennigan, E. Noland, K. Millican, G. van den Driessche, B. Damoc, A. Guy, S. Osindero, K. Simonyan, E. Elsen, J. W. Rae, O. Vinyals and L. Sifre, *Training compute-optimal large language models*, 2022. arXiv: `2203.15556 [cs.CL]`.

[23] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. Rao, P. Barnes, Y. Tay, N. Shazeer, V. Prabhakaran, E. Reif, N. Du, B. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. Garcia, V. Misra, K. Robinson, L. Fedus, D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi, D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai, M. Pellat, A. Lewkowycz, E. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Diaz, O. Firat, M. Catasta, J. Wei, K. Meier-Hellstern, D. Eck, J. Dean, S. Petrov and N. Fiedel, *Palm: Scaling language modeling with pathways*, 2022. arXiv: `2204.02311 [cs.CL]`.

[24] J. Devlin, M.-W. Chang, K. Lee and K. Toutanova, *Bert: Pre-training of deep bidirectional transformers for language understanding*, 2018. arXiv: `1810.04805 [cs.CL]`.

[25] I. Tenney, D. Das and E. Pavlick, ‘Bert rediscovers the classical nlp pipeline,’ in *Association for Computational Linguistics*, 2019. [Online]. Available: `https://arxiv.org/abs/1905.05950`.

[26] OpenAI, *Openai’s chatgpt*, `https://chat.openai.com/`, Accessed: 2023-05-10.

[27] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever *et al.*, ‘Improving language understanding by generative pre-training,’ 2018.

[28] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, ‘Language models are unsupervised multitask learners,’ *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[29] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever and D. Amodei, *Language models are few-shot learners*, 2020. arXiv: `2005.14165 [cs.CL]`.

[30] A. Wang, Y. Pruksachatkun, N. Nangia, A. Singh, J. Michael, F. Hill, O. Levy and S. Bowman, ‘Superglue: A stickier benchmark for general-purpose language understanding systems,’ in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox and R. Garnett, Eds., vol. 32, Curran Associates, Inc., 2019. [Online]. Avail-

able: https://proceedings.neurips.cc/paper_files/paper/2019/file/4496bf24afe7fab6f046bf4923da8de6-Paper.pdf.

[31] OpenAI, *Model index for researchers*, https://platform.openai.com/docs/model-index-for-researchers, Accessed: 2023-05-21.

[32] J. Buolamwini, *When the robot doesn't see dark skin*, https://www.nytimes.com/2018/06/21/opinion/facial-analysis-technology-bias.html, Accessed: 2023-05-10.

[33] P. Verma, *These robots were trained on ai. they became racist and sexist.* https://www.washingtonpost.com/technology/2022/07/16/racist-robots-ai/, Accessed: 2023-05-10.

[34] OpenAI, *Aligning language models to follow instructions*, https://openai.com/research/instruction-following, Accessed: 2023-05-21.

[35] S. Lin, J. Hilton and O. Evans, *Truthfulqa: Measuring how models mimic human falsehoods*, 2022. arXiv: 2109.07958 [cs.CL].

[36] S. Gehman, S. Gururangan, M. Sap, Y. Choi and N. A. Smith, *Realtoxicityprompts: Evaluating neural toxic degeneration in language models*, 2020. arXiv: 2009.11462 [cs.CL].

[37] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. Christiano, J. Leike and R. Lowe, *Training language models to follow instructions with human feedback*, 2022. arXiv: 2203.02155 [cs.CL].

[38] K. Hu, *Chatgpt sets record for fastest-growing user base - analyst note, 02.02.2023*, https://www.reuters.com/technology/chatgpt-sets-record-fastest-growing-user-base-analyst-note-2023-02-01/, Accessed: 2023-05-21.

[39] OpenAI, *Openai api reference*, https://platform.openai.com/docs/api-reference, Accessed: 2023-05-21.

[40] OpenAI, *Openai codex*, https://openai.com/blog/openai-codex, Accessed: 2023-05-29.

[41] Microsoft, *Github home page*, https://github.com/, Accessed: 2023-03-29.

[42] Github, *Github copilot home page*, https://github.com/features/copilot, Accessed: 2023-03-29.

[43] OpenAI, *Gpt-4 technical report*, 2023. arXiv: 2303.08774 [cs.CL].

[44] OpenAI, *Openai gpt-4 blog post*, https://openai.com/gpt-4, Accessed: 2023-06-02.

[45] OpenAI, *Openai gpt-4 research blog post*, https://openai.com/research/gpt-4, Accessed: 2023-06-02.

[46] A. R. Hevner, S. T. March, J. Park and S. Ram, 'Design science in information systems research,' *MIS Quarterly*, vol. 28, no. 1, pp. 75–105, 2004, ISSN: 02767783. [Online]. Available: `http://www.jstor.org/stable/25148625` (visited on 11/03/2023).

[47] Refactoring.Guru, *Builder*, `https://refactoring.guru/design-patterns/builder`, Accessed: 2023-06-09.

[48] Atlassian, *What is scrum*, `https://www.atlassian.com/agile/scrum`, Accessed: 2023-06-01.

[49] Gitlab, *Gitlab about page*, `https://about.gitlab.com/`, Accessed: 2023-06-02.

[50] M. Lacchia, *Radon home page*, `https://radon.readthedocs.io/en/latest/`, Accessed: 2023-04-18.

[51] Unknown, *W3resource home page*, `https://www.w3resource.com/`, Accessed: 2023-04-17.

[52] S. Sarsa, P. Denny, A. Hellas and J. Leinonen, 'Automatic generation of programming exercises and code explanations using large language models,' in *Proceedings of the 2022 ACM Conference on International Computing Education Research V.1*, ACM, Aug. 2022. DOI: `10.1145/3501385.3543957`. [Online]. Available: `https://doi.org/10.1145%2F3501385.3543957`.

[53] J. Finnie-Ansley, P. Denny, B. Becker, A. Luxton-Reilly and J. Prather, 'The robots are coming: Exploring the implications of openai codex on introductory programming,' Feb. 2022, pp. 10–19. DOI: `10.1145/3511861.3511863`.

[54] E. Soloway, 'Learning to program = learning to construct mechanisms and explanations,' *Commun. ACM*, vol. 29, no. 9, pp. 850–858, Sep. 1986, ISSN: 0001-0782. DOI: `10.1145/6592.6594`. [Online]. Available: `https://doi.org/10.1145/6592.6594`.

[55] Simon, 'Soloway's rainfall problem has become harder,' in *Proceedings of the 2013 Learning and Teaching in Computing and Engineering*, ser. LATICE '13, USA: IEEE Computer Society, 2013, pp. 130–135, ISBN: 9780769549606. DOI: `10.1109/LaTiCE.2013.44`. [Online]. Available: `https://doi.org/10.1109/LaTiCE.2013.44`.

[56] K. Fisler, 'The recurring rainfall problem,' in *Proceedings of the Tenth Annual Conference on International Computing Education Research*, ser. ICER '14, Glasgow, Scotland, United Kingdom: Association for Computing Machinery, 2014, pp. 35–42, ISBN: 9781450327558. DOI: `10.1145/2632320.2632346`. [Online]. Available: `https://doi.org/10.1145/2632320.2632346`.

[57] A. Ebrahimi, 'Novice programmer errors: Language constructs and plan composition,' *International Journal of Human-Computer Studies*, vol. 41, no. 4, pp. 457–480, 1994, ISSN: 1071-5819. DOI: `https://doi.org/10.1006/ijhc.1994.1069`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S107158198471069X`.

[58] M. Guzdial, R. Fithian, A. Forte and L. Rich, *Report on pilot offering of cs1315 introduction to media computation with comparison to cs1321 and coe1361*, 2003.

[59] A.-J. Lakanen, V. Lappalainen and V. Isomöttönen, 'Revisiting rainfall to explore exam questions and performance on cs1,' in *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, ser. Koli Calling '15, Koli, Finland: Association for Computing Machinery, 2015, pp. 40–49, ISBN: 9781450340205. DOI: `10.1145/2828959.2828970`. [Online]. Available: `https://doi.org/10.1145/2828959.2828970`.

[60] React.dev, *React*, `https://react.dev/`, Accessed: 2023-05-10.

[61] Inspera, *Inspera assessment*, `https://www.inspera.com/assessment`, Accessed: 2023-06-09.

[62] Pallets, *Flask*, `https://flask.palletsprojects.com/en/2.3.x/`, Accessed: 2023-05-10.

[63] MongoDB, *Mongodb atlas database*, `https://www.mongodb.com/atlas/database`, Accessed: 2023-05-10.

[64] J. Sæther and J. Mæland, *Taskmaster gitlab*, `https://gitlab.com/Jonassat/taskmaster-remaster`, Accessed: 2023-05-15.

Appendix A

Additional Material

The reference exercise set from the course ITGK, along with generated exercise sets and solutions used in the student demonstrations are attached here.

Demo 26.05 1

NTNU
Kunnskap for en bedre verden

Generert: 26 May, 2023 18:58:10

Poeng: 120

Oppgave 1. 5 poeng

Skriv en funksjon som tar inn to input fra brukeren: et adjektiv og et tall. Funksjonen skal deretter skrive ut en setning som inneholder adjektivet og en multiplikator, som øker med 1 for hver iterasjon, opp til og med det gitte tallet. Setningen skal være på formatet "I am X times more Y!", der X er multiplikatoren og Y er adjektivet.

Oppgave 2. 5 poeng

Lag et program som ber brukeren om å skrive inn adjektiver. For hvert adjektiv brukeren skriver inn, skal programmet skrive ut en setning som sier "I am more [adjektiv]!". Programmet skal fortsette å be om adjektiver og skrive ut setninger til brukeren ikke skriver inn noe og trykker enter.

Oppgave 3. 5 poeng

Skriv et program som ber brukeren om å skrive inn adjektiver. Programmet skal fortsette å be om adjektiver og skrive ut en setning som inneholder det innskrevne adjektivet, for eksempel "I am more [adjective]!". Programmet skal stoppe når det har mottatt minst 42 adjektiver eller når den totale lengden av alle adjektivene som er skrevet inn overstiger 42 tegn.

Oppgave 4. 5 poeng

Skriv en funksjon som gjør følgende:

1. Skriv ut en melding som sier "5-multiplication table between 20 and 81:".
2. Skriv ut alle tallene i 5-gangen mellom 20 og 81 (inkludert) på samme linje, separert med mellomrom.
3. Skriv ut en melding som sier "Numbers from 48 to 80 incrementing by 8:".
4. Skriv ut alle tallene fra 48 til 80 (inkludert) med en økning på 8 mellom hvert tall, på samme linje, separert med mellomrom.
5. Skriv ut en melding som sier "Numbers backwards from 100 to 80 incrementing by -3:".
6. Skriv ut alle tallene fra 100 til 80 (inkludert) med en økning på -3 mellom hvert tall, på samme linje, separert med mellomrom.

Eksempel på kjøring:

```
print_numbers()
>>> 5-multiplication table between 20 and 81:
20 25 30 35 40 45 50 55 60 65 70 75 80
Numbers from 48 to 80 incrementing by 8:
48 56 64 72 80
Numbers backwards from 100 to 80 incrementing by -3:
100 97 94 91 88 85 82
```

Oppgave 5. 5 poeng

Skriv en funksjon som heter "print_numbers" som skriver ut alle heltall fra 1 til 5 (inkludert) ved hjelp av en while-løkke. Funksjonen skal ikke ta inn noen parametere og skal ikke returnere noe.

Eksempel på kjøring:

```
print_numbers()
>>> 1
2
3
4
5
```

Oppgave 6. 5 poeng

Skriv en funksjon som tar inn et heltall som argument (standardverdi 15) og skriver ut alle heltall fra dette tallet og ned til 1 i synkende rekkefølge. Implementer funksjonen ved hjelp av rekursjon.

Eksempel på kjøring:

```
print_numbers_recursion(15)
>>> 15
14
13
12
11
10
..
4
3
2
1
```

Oppgave 7. 5 poeng

Skriv et program som tar inn 7 heltall fra brukeren, legger sammen alle tallene og skriver ut summen. Bruk liste-forståelse for å samle inn tallene fra brukeren.

Oppgave 8. 5 poeng

Skriv en rekursiv funksjon som multipliserer sammen alle tallene fra 1 og oppover, og returnerer produktet så snart det overstiger 1000. Funksjonen skal ikke ta inn noen argumenter.

Eksempel på kjøring:

```
print(multiply_recursive(1, 1))
>>> 5040
```

Oppgave 9. 5 poeng

Skriv et program som spør brukeren om hovedstaden i Niue. Programmet skal fortsette å spørre brukeren inntil riktig svar er gitt. Når brukeren gir riktig svar, skal programmet skrive ut en melding som bekrefter at svaret er korrekt, samt antall forsøk det tok for brukeren å gjette riktig. Programmet skal ikke være case-sensitive, det vil si at det skal akseptere svar uavhengig av store og små bokstaver.

Oppgave 10. 5 poeng

Skriv et program som tar inn to tall: antall studenter og antall fag. Programmet skal deretter generere og skrive ut alle mulige kombinasjoner av studenter og fag, der hver student elsker hvert fag. For eksempel, hvis det er 2 studenter og 3 fag, skal programmet skrive ut følgende:

```
Student 1 loves subject 1
Student 1 loves subject 2
Student 1 loves subject 3
Student 2 loves subject 1
Student 2 loves subject 2
Student 2 loves subject 3
```

Oppgave 11. 5 poeng

Skriv et program som genererer og skriver ut alle mulige tidspunkt i et døgn med formatet "HH:mm", hvor HH representerer timer (00-23) og mm representerer minutter (00-59).

Eksempel på kjøring:

```
print_timestamps_v()
```

```
>>> 00:00
00:01
00:02
00:03
00:04
00:05
..
23:56
23:57
23:58
23:59
```

Oppgave 12. 5 poeng

Skriv en funksjon som genererer og skriver ut multiplikasjonstabellen for tallene fra 1 til 10. Funksjonen skal skrive ut hvert produkt på en egen linje og ha en tom linje mellom hver tabell.

Eksempel på kjøring:

```
multiplication_tables()
>>> 1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
1 * 5 = 5
1 * 6 = 6
..
10 * 9 = 90
10 * 10 = 100
```

Oppgave 13. 5 poeng

Hva er output av følgende kjøring?

```
def floor_divide(n=345):
    result = ""
    for _ in range(n):
        n //= 2
        result += str(n) + " "
        if n == 0:
            break
    return result.strip()

print(floor_divide(345))
```

Oppgave 14. 5 poeng

Hva er output av følgende kjøring?

```
def print_even_numbers_v():
    for i in range(0, 11, 2):
        if i % 4 == 0:
            print(i, "divisible by 4")
        else:
            print(i)

print_even_numbers_v()
```

Oppgave 15. 5 poeng

Hva er output av følgende kjøring?

```
def multiply_with_for_loop():
    result = 1
    for _ in range(10):
        result *= 2
    print(result)

multiply_with_for_loop()
```

Oppgave 16. 5 poeng

Hva er output av følgende kjøring?

```
def multiply_with_while_loop():
    value = 1
    count = 0
    while count < 3:
        value *= 2
        count += 1
    print(value)

multiply_with_while_loop()
```

Oppgave 17. 5 poeng

Hva er output av følgende kjøring?

```
def print_stars_list_comprehension(i=5):
```

```
    [print('*' * (x + 1)) for x in range(i)]

print_stars_list_comprehension(5)
```

Oppgave 18. 5 poeng

Skriv et program som genererer et tilfeldig tall mellom to grenser gitt av brukeren. Programmet skal deretter be brukeren om å gjette tallet. Hvis brukerens gjetning er lavere enn det tilfeldige tallet, skal programmet gi beskjed om å gjette høyere. Hvis gjetningen er høyere, skal programmet gi beskjed om å gjette lavere. Når brukeren gjetter riktig tall, skal programmet gratulere brukeren og avslutte. Hvis brukeren gir ugyldig input (ikke et tall), skal programmet be om et nytt tall og informere om at input er ugyldig.

Oppgave 19. 5 poeng

Skriv en funksjon som beregner summen av en geometrisk rekke ved hjelp av rekursjon. Funksjonen skal ta inn to parametere: n, som er antall ledd i rekken, og r, som er det konstante forholdet mellom hvert ledd i rekken (standardverdien for r skal være 2). Funksjonen skal returnere summen av rekken.

Eksempel på kjøring:

```
print(geometric_sum_recursive(24, 2))
>>> 33554431
```

Oppgave 20. 5 poeng

Skriv en funksjon som tar inn et heltall n og returnerer det n-te tallet i Fibonacci-sekvensen ved hjelp av rekursjon. Fibonacci-sekvensen er en sekvens av tall der hvert tall er summen av de to foregående tallene, med de to første tallene i sekvensen som 0 og 1.

Eksempel på kjøring:

```
print(fibonacci_recursive(10))
>>> 55
```

Oppgave 21. 5 poeng

Skriv en funksjon som genererer de første n Fibonacci-tallene. Funksjonen skal ta inn et heltall n som argument og returnere en generator som gir Fibonacci-tallene i rekkefølge. Fibonacci-sekvensen starter med tallene 0 og 1, og hvert påfølgende tall er summen av de to foregående tallene (0, 1, 1, 2, 3, 5, 8, ...).

Eksempel på kjøring:

```
print(fibonacci_generator(34))
>>>
```

Oppgave 22. 5 poeng

Skriv en funksjon som tar inn et heltall 'n' og returnerer en liste med alle Fibonacci-tallene som er mindre eller lik 'n'. Fibonacci-tallene er en sekvens av tall der hvert tall er summen av de to foregående tallene, med de to første tallene i sekvensen som 0 og 1.

Eksempel på kjøring:

```
print(fibonacci_loop(33))
>>> [0, 1, 1, 2, 3, 5, 8, 13, 21]
```

Oppgave 23. 5 poeng

Skriv en funksjon som beregner summen av en tallserie, der hvert element i serien er kvadratet av sin indeks, og har alternerende fortegn. Serien starter med indeks 1 og går opp til og med et gitt heltall n. Funksjonen skal ta inn et heltall n som argument og returnere summen av serien.

Eksempel på kjøring:

```
print(sum_of_series(23))
>>> -276
```

Oppgave 24. 5 poeng

Lag et enkelt Hangman-spill

Du skal lage et enkelt Hangman-spill i Python. Spillet skal ta inn to parametere: et ord som skal gjettes og et maksimalt antall forsøk. Spillet skal fortsette til enten ordet er gjettet eller antall forsøk er brukt opp. For hvert forsøk skal brukeren gjette en bokstav. Hvis bokstaven er i ordet, skal den vises på riktig plass i det gjette ordet. Hvis ikke, skal antall forsøk reduseres med \xe9n. Etter hvert forsøk skal det gjette ordet vises med riktig gjette bokstaver og ubrukte plasser representert med understrek (_). Hvis ordet blir gjettet før antall forsøk er brukt opp, skal spillet gratulere brukeren med seier. Hvis ikke, skal spillet informere brukeren om at de har tapt og vise det riktige ordet.

Demo 26.05 2

NTNU
Kunnskap for en bedre verden

Generert: 26 May, 2023 18:58:12

Poeng: 120

Oppgave 1. 5 poeng

Lag et program som tar inn et adjektiv og et tall fra brukeren. Programmet skal deretter generere og skrive ut en liste med setninger som inneholder adjektivet og en økende multiplikator. For eksempel, hvis brukeren skriver inn adjektivet "happy" og tallet 3, skal programmet skrive ut følgende setninger:

1. I am 1 times more happy!
2. I am 2 times more happy!
3. I am 3 times more happy!

Oppgave 2. 5 poeng

Lag et program som ber brukeren om å skrive inn adjektiver. For hvert adjektiv brukeren skriver inn, skal programmet skrive ut en setning som sier "I am more [adjektiv]!". Programmet skal fortsette å be om adjektiver og skrive ut setninger til brukeren ikke skriver inn noe og trykker enter.

Oppgave 3. 5 poeng

Skriv et program som ber brukeren om å skrive inn adjektiver. Programmet skal fortsette å be om adjektiver og skrive ut en setning som sier "I am more [adjektiv]!" for hvert adjektiv brukeren skriver inn. Programmet skal stoppe når den totale lengden av alle adjektivene som er skrevet inn overstiger 42 tegn.

Oppgave 4. 5 poeng

Skriv en funksjon som gjør følgende:

1. Skriv ut alle tallene i 5-gangen mellom 20 og 81 (inkludert).
2. Skriv ut alle tallene fra 48 til 80 (inkludert) med en økning på 8.
3. Skriv ut alle tallene fra 100 til 80 (inkludert) i synkende rekkefølge med en økning på -3.

Eksempel på kjøring:

```
print_numbers()
>>> 5-multiplication table between 20 and 81:
20 25 30 35 40 45 50 55 60 65 70 75 80
Numbers from 48 to 80 incrementing by 8:
48 56 64 72 80
Numbers backwards from 100 to 80 incrementing by -3:
100 97 94 91 88 85 82
```

Oppgave 5. 5 poeng

Skriv en funksjon som heter "print_numbers" som skriver ut alle heltall fra 1 til 5, inkludert begge endepunktene. Funksjonen skal ikke ta inn noen parametere og skal bruke en for-løkke for å iterere gjennom tallene.

Eksempel på kjøring:

```
print_numbers()
>>> 1
2
3
4
5
```

Oppgave 6. 5 poeng

Skriv en funksjon som heter "print_numbers_for_range" som ikke tar noen argumenter. Når funksjonen kalles, skal den skrive ut tallene fra 15 til 1 i synkende rekkefølge, hver på en ny linje.

Eksempel på kjøring:

```
print_numbers_for_range()
>>> 15
14
13
12
11
10
..
4
3
2
1
```

Oppgave 7. 5 poeng

Skriv en funksjon som tar inn 7 heltall fra brukeren ved hjelp av en for-løkke, og deretter beregner og skriver ut summen av disse tallene.

Oppgave 8. 5 poeng

Skriv en funksjon som multipliserer sammen tallene fra 1 til 1000 ved hjelp av en iterator. Funksjonen skal stoppe multiplikasjonen og returnere produktet så snart det overstiger 1000.

Eksempel på kjøring:

```
print(multiply_for_iterator())
>>> 5040
```

Oppgave 9. 5 poeng

Skriv et program som spør brukeren om hovedstaden i Niue. Programmet skal fortsette å spørre brukeren inntil riktig svar er gitt. Når brukeren gir riktig svar, skal programmet skrive ut en melding som bekrefter at svaret er korrekt, samt antall forsøk det tok for brukeren å gjette riktig. Programmet skal ikke være case-sensitive, det vil si at det skal akseptere svar uavhengig av store og små bokstaver.

Oppgave 10. 5 poeng

Skriv et program som tar inn antall studenter og antall fag som input. Programmet skal deretter generere og skrive ut alle mulige kombinasjoner av studenter og fag, og vise hvilken student som elsker hvilket fag. For eksempel, hvis det er 2 studenter og 3 fag, skal programmet skrive ut følgende:

```
Student 1 loves subject 1
Student 1 loves subject 2
Student 1 loves subject 3
Student 2 loves subject 1
Student 2 loves subject 2
Student 2 loves subject 3
```

Oppgave 11. 5 poeng

Skriv en funksjon som genererer og skriver ut alle mulige tidspunkt i et døgn med 24 timer og 60 minutter, i formatet "HH:mm". Tidspunktene skal skrives ut i stigende rekkefølge, og hver time og hvert minutt skal ha to siffer, med eventuelle ledende nuller.

Eksempel på kjøring:

```
print_timestamps_v()
>>> 00:00
```

```
00:01
00:02
00:03
00:04
00:05
..
23:56
23:57
23:58
23:59
```

Oppgave 12. 5 poeng

Skriv en funksjon som genererer og skriver ut multiplikasjonstabellen for tallene fra 1 til 10. Hver linje skal vise et multiplikasjonsuttrykk og resultatet, og det skal være et linjeskift mellom hver tabell.

Eksempel på kjøring:

```
multiplication_tables()
>>> 1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
1 * 5 = 5
1 * 6 = 6
..
10 * 9 = 90
10 * 10 = 100
```

Oppgave 13. 5 poeng

Hva er output av følgende kjøring?

```
def floor_divide(n=345):
    result = ""
    while n > 0:
        n //= 2
        result += str(n) + " "
    return result.strip()

print(floor_divide(345))
```

Oppgave 14. 5 poeng

Hva er output av følgende kjøring?

```
def print_even_numbers_v():
    even_numbers = [i for i in range(11) if i % 2 == 0]
    for num in even_numbers:
        if num % 4 == 0:
            print(num, "divisible by 4")
        else:
            print(num)

print_even_numbers_v()
```

Oppgave 15. 5 poeng

Hva er output av følgende kjøring?

```
def multiply_with_recursion(result=1, count=0):
    if count == 10:
        print(result)
    else:
        multiply_with_recursion(result * 2, count + 1)

multiply_with_recursion(1, 0)
```

Oppgave 16. 5 poeng

Hva er output av følgende kjøring?

```
def multiply_with_recursion(value=1, count=3):
    if count == 0:
        print(value)
    else:
        multiply_with_recursion(value * 2, count - 1)

multiply_with_recursion(1, 3)
```

Oppgave 17. 5 poeng

Hva er output av følgende kjøring?

```
def print_stars_while_loop(i=5):
    x = 0
    while x < i:
        print('*' * (x + 1))
```

```
    x += 1

print_stars_while_loop(5)
```

Oppgave 18. 5 poeng

Lag et enkelt tallgjettingsspill der brukeren skal gjette et tilfeldig tall mellom to grenser. Programmet skal be brukeren om å angi en nedre og øvre grense for tallet. Deretter skal programmet generere et tilfeldig tall mellom disse grensene og be brukeren om å gjette tallet. For hver gjetning skal programmet gi tilbakemelding om brukeren skal gjette høyere eller lavere. Når brukeren gjetter riktig tall, skal programmet gratulere brukeren og avslutte spillet.

Oppgave 19. 5 poeng

Skriv en funksjon som beregner summen av en geometrisk rekke med n ledd og en gitt felles ratio r. Funksjonen skal ta inn to parametere: n, som representerer antall ledd i rekken, og r, som representerer den felles ratioen (standardverdien for r skal være 2). Funksjonen skal returnere summen av rekken. Du skal bruke list comprehension for å generere rekken og beregne summen.

Eksempel på kjøring:

```
print(geometric_sum_list_comprehension(33, 2))
>>> 17179869183
```

Oppgave 20. 5 poeng

Skriv en funksjon som beregner det n-te Fibonacci-tallet ved hjelp av en iterativ metode. Funksjonen skal ta inn et heltall n (0 \u2264 n \u2264 10^6) og returnere det n-te Fibonacci-tallet. Fibonacci-sekvensen starter med 0 og 1, og hvert påfølgende tall er summen av de to foregående tallene (0, 1, 1, 2, 3, 5, 8, ...).

Eksempel på kjøring:

```
print(fibonacci_iterative(33))
>>> 3524578
```

Oppgave 21. 5 poeng

Skriv en funksjon som beregner summen av Fibonacci-tallene opp til og med det n-te Fibonacci-tallet ved hjelp av rekursjon. Funksjonen skal ta inn et heltall n som argument og returnere summen av Fibonacci-tallene opp til og med det n-te tallet. Hvis n er mindre enn eller lik 0, skal funksjonen returnere 0. Hvis n er lik 1, skal funksjonen returnere 1.

Eksempel på kjøring:

```
print(fibonacci_sum_recursive(16))
>>> 2583
```

Oppgave 22. 5 poeng

Skriv en funksjon som tar inn et heltall 'n' og returnerer en liste med de første 'n' Fibonacci-tallene. For å forbedre ytelsen, bruk memoisering for å lagre tidligere beregnede resultater og unngå unødvendige gjentakelser.

Eksempel på kjøring:

```
print(fibonacci_memo(7, {}))
>>> [0, 1, 1, 2, 3, 5, 8]
```

Oppgave 23. 5 poeng

Skriv en funksjon som tar inn et heltall n og returnerer summen av en tallserie der hvert oddetall i serien er kvadrert og lagt til totalen, mens hvert partall i serien er kvadrert og trukket fra totalen. Serien skal starte fra 1 og gå opp til og med n.

Eksempel på kjøring:

```
print(sum_of_series(14))
>>> -105
```

Oppgave 24. 5 poeng

Lag et enkelt Hangman-spill i Python. Spillet skal ta inn to parametere: et ord som skal gjettes og et maksimalt antall forsøk. Spillet skal be brukeren om å gjette en bokstav om gangen og vise gjeldende status for det gjette ordet ved å erstatte ukjente bokstaver med understrek (_). Hvis brukeren gjetter riktig bokstav, skal den vises på riktig plass i ordet. Hvis brukeren gjetter feil, skal antall forsøk reduseres med \xe9n. Spillet fortsetter til enten brukeren har gjettet ordet eller brukt opp alle forsøkene. Til slutt skal spillet vise en melding om brukeren vant eller tapte, og avsløre det riktige ordet.

Demo 26.05 3

Generert: 26 May, 2023 18:58:14

Poeng: 120

Oppgave 1. 5 poeng

Lag et rekursivt program som tar inn et adjektiv og et tall som input fra brukeren. Programmet skal deretter skrive ut en setning som sier "I am X times more [adjective]!", hvor X er et tall fra 1 til det gitte tallet, og [adjective] er det innskrevne adjektivet.

Oppgave 2. 5 poeng

Lag et rekursivt program som kontinuerlig ber brukeren om å skrive inn et adjektiv. For hvert adjektiv som blir skrevet inn, skal programmet skrive ut en setning som sier "I am more [adjektiv]!". Programmet skal avsluttes når brukeren ikke skriver inn noe og trykker enter.

Oppgave 3. 5 poeng

Skriv en funksjon som kontinuerlig ber brukeren om å skrive inn adjektiver. For hvert adjektiv som blir skrevet inn, skal funksjonen legge til lengden av adjektivet til en total sum av tegn. Funksjonen skal fortsette å be om adjektiver inntil summen av tegn er lik eller større enn 42. For hvert adjektiv som blir skrevet inn, skal funksjonen også skrive ut en setning som sier "I am more [adjective]!", hvor [adjective] er det innskrevne adjektivet. Hvis det oppstår en feil underveis, skal funksjonen håndtere dette og fortsette å be om adjektiver.

Oppgave 4. 5 poeng

Skriv en funksjon som gjør følgende:

1. Skriv ut 5-gangen mellom 20 og 81 (inkludert) på \xe9n linje.
2. Skriv ut tallene fra 48 til 80 (inkludert) med en økning på 8 mellom hvert tall på en ny linje.
3. Skriv ut tallene fra 100 til 80 (inkludert) med en reduksjon på 3 mellom hvert tall på en tredje linje.

Funksjonen skal ikke ta inn noen argumenter og skal ikke returnere noe.

Eksempel på kjøring:

```
print_numbers()
>>> 5-multiplication table between 20 and 81:
20 25 30 35 40 45 50 55 60 65 70 75 80
Numbers from 48 to 80 incrementing by 8:
48 56 64 72 80
Numbers backwards from 100 to 80 incrementing by -3:
100 97 94 91 88 85 82
```

Oppgave 5. 5 poeng

Skriv en funksjon som skriver ut tallene fra 1 til 5 (inkludert) ved hjelp av list comprehension. Fungerende kode skal kunne kjøres uten å ta inn noen parametere og skal ikke returnere noe, kun skrive ut tallene.

Eksempel på kjøring:

```
print_numbers_list_comprehension()
>>> 1
2
3
4
5
```

Oppgave 6. 5 poeng

Skriv en funksjon som heter "print_numbers" som ikke tar noen argumenter. Når funksjonen kalles, skal den skrive ut tallene fra 15 til 1 i synkende rekkefølge, ett tall per linje. Bruk en while-løkke for å løse oppgaven.

Eksempel på kjøring:

```
print_numbers()
>>> 15
14
13
12
11
10
..
4
3
2
1
```

Oppgave 7. 5 poeng

Skriv et program som ber brukeren om å taste inn 7 heltall. Programmet skal deretter beregne og skrive ut summen av disse tallene. Du kan benytte deg av funksjonene `input()`, `int()`, `sum()` og `map()` for å løse oppgaven.

Oppgave 8. 5 poeng

Skriv en funksjon som finner det minste tallet som kan multipliseres med alle heltall fra 1 og oppover, slik at produktet blir større enn 1000. Funksjonen skal returnere dette produktet.

Eksempel på kjøring:

```
print(multiply())
>>> 5040
```

Oppgave 9. 5 poeng

Du skal lage et enkelt quiz-spill som spør brukeren om hovedstaden i Niue. Brukeren får ubegrenset antall forsøk, men programmet skal holde styr på hvor mange forsøk brukeren har brukt. Når brukeren svarer riktig, skal programmet gratulere brukeren, oppgi hovedstaden i Niue og antall forsøk som ble brukt. Programmet skal ikke være case-sensitive, det vil si at det skal akseptere svar uavhengig av store og små bokstaver.

Oppgave 10. 5 poeng

Skriv et program som tar inn antall studenter og antall fag som input. Programmet skal deretter generere og skrive ut alle mulige kombinasjoner av studenter og fag, hvor hver kombinasjon viser hvilken student som elsker hvilket fag. Bruk en funksjon for å implementere dette.

Oppgave 11. 5 poeng

Skriv en funksjon som genererer og skriver ut alle mulige tidspunkt i et døgn med formatet "HH:mm", der HH representerer timer (00-23) og mm representerer minutter (00-59).

Eksempel på kjøring:

```
print_timestamps_v()
>>> 00:00
00:01
00:02
00:03
00:04
00:05
..
23:56
```

```
23:57
23:58
23:59
```

Oppgave 12. 5 poeng

Skriv en funksjon som genererer og skriver ut multiplikasjonstabellen for tallene fra 1 til 10. Hver linje skal vise et multiplikasjonsuttrykk og resultatet, og det skal være en tom linje mellom hver tabell.

Eksempel på kjøring:

```
multiplication_tables()
>>> 1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
1 * 5 = 5
1 * 6 = 6
..
10 * 9 = 90
10 * 10 = 100
```

Oppgave 13. 5 poeng

Hva er output av følgende kjøring?

```
def floor_divide(n=345):
    result = ""
    for _ in range(n):
        n //= 2
        result += str(n) + " "
        if n == 0:
            break
    return result.strip()

print(floor_divide(345))
```

Oppgave 14. 5 poeng

Hva er output av følgende kjøring?

```
def print_even_numbers_v():
    i = 0
    while i <= 10:
```

```
        if i % 4 == 0:
            print(i, "divisible by 4")
        else:
            print(i)
        i += 2

print_even_numbers_v()
```

Oppgave 15. 5 poeng

Hva er output av følgende kjøring?

```
def multiply_with_for_loop():
    result = 1
    for _ in range(10):
        result *= 2
    print(result)

multiply_with_for_loop()
```

Oppgave 16. 5 poeng

Hva er output av følgende kjøring?

```
def multiply_with_for_loop():
    value = 1
    for _ in range(3):
        value *= 2
    print(value)

multiply_with_for_loop()
```

Oppgave 17. 5 poeng

Hva er output av følgende kjøring?

```
def print_stars_for_loop(i=5):
    for x in range(i):
        print('*' * (x + 1))

print_stars_for_loop(5)
```

Oppgave 18. 5 poeng

Lag et enkelt tallgjettingsspill der brukeren skal gjette et tilfeldig tall mellom to grenser. Programmet skal be brukeren om å oppgi en nedre og en øvre grense for tallet. Deretter skal programmet generere et tilfeldig tall mellom disse grensene og be brukeren om å gjette tallet. For hver gjetning skal programmet gi tilbakemelding om brukeren skal gjette høyere eller lavere. Når brukeren gjetter riktig tall, skal programmet gratulere brukeren og avslutte spillet.

Oppgave 19. 5 poeng

Skriv en funksjon som beregner summen av en geometrisk rekke. Funksjonen skal ta inn to parametere: n og r. n er antall ledd i rekken, og r er det konstante forholdet mellom hvert ledd i rekken. Hvis r ikke er gitt, skal det antas å være 2. Funksjonen skal returnere summen av rekken.

Eksempel på kjøring:

```
print(geometric_sum(25, 2))
>>> 67108863
```

Oppgave 20. 5 poeng

Skriv en funksjon som tar inn et heltall n og returnerer det n-te tallet i Fibonacci-sekvensen ved hjelp av rekursjon. Fibonacci-sekvensen er en sekvens av tall der hvert tall er summen av de to foregående tallene, med de to første tallene i sekvensen som 0 og 1.

Eksempel på kjøring:

```
print(fibonacci_recursive(10))
>>> 55
```

Oppgave 21. 5 poeng

Skriv en funksjon som tar inn et heltall n og returnerer summen av de første n Fibonacci-tallene. Hvis n er mindre eller lik 0, skal funksjonen returnere 0.

Eksempel på kjøring:

```
print(fibonacci_sum(14))
>>> 609
```

Oppgave 22. 5 poeng

Skriv en funksjon som tar inn et heltall 'n' og returnerer en liste med alle Fibonacci-tallene som er mindre eller lik 'n'. Fibonacci-tallene er en sekvens av tall der hvert tall er summen av de to foregående tallene, med de to første tallene i sekvensen som 0 og 1.

Eksempel på kjøring:

```
print(fibonacci_loop(33))
>>> [0, 1, 1, 2, 3, 5, 8, 13, 21]
```

Oppgave 23. 5 poeng

Skriv en funksjon som beregner summen av en tallserie, der hvert element i serien er kvadratet av indeksen multiplisert med -1 opphøyd i indeksen modulo 2. Serien starter med indeks 1 og går opp til og med et gitt heltall n. Funksjonen skal ta inn ett argument, n, og returnere summen av serien.

Eksempel på kjøring:

```
print(sum_of_series(9))
>>> -45
```

Oppgave 24. 5 poeng

Lag et enkelt Hangman-spill

Du skal lage et enkelt Hangman-spill i Python. Spillet skal ta inn to parametere: et ord som skal gjettes og et maksimalt antall forsøk. Spillet skal fortsette til enten ordet er gjettet eller antall forsøk er brukt opp. For hvert forsøk skal brukeren gjette en bokstav. Hvis bokstaven er i ordet, skal den vises på riktig plass i det gjette ordet. Hvis ikke, skal antall forsøk reduseres med \xe9n. Etter hvert forsøk skal det gjette ordet vises med riktig gjette bokstaver og understreker for ikke-gjette bokstaver. Hvis ordet blir gjettet før antall forsøk er brukt opp, skal spillet gratulere brukeren med seier. Hvis ikke, skal spillet informere brukeren om at de har tapt og vise det riktige ordet.

Demo 26.05 4

NTNU
Kunnskap for en bedre verden

Generert: 26 May, 2023 18:58:16

Poeng: 120

Oppgave 1. 5 poeng

Skriv et program som tar inn to input fra brukeren: et adjektiv og et heltall. Programmet skal deretter skrive ut en setning som inneholder adjektivet og en multiplikator, som øker med 1 for hver iterasjon, opp til og inkludert det gitte heltallet. Setningen skal være på formatet "I am X times more ADJECTIVE!", der X er multiplikatoren og ADJECTIVE er det gitte adjektivet.

Oppgave 2. 5 poeng

Lag et program som kontinuerlig ber brukeren om å skrive inn adjektiver. For hvert adjektiv som blir skrevet inn, skal programmet skrive ut en setning som sier "I am more [adjektiv]!". Programmet skal avsluttes når brukeren ikke skriver inn noe og bare trykker på enter-tasten.

Oppgave 3. 5 poeng

Skriv et program som kontinuerlig ber brukeren om å skrive inn adjektiver. For hvert adjektiv som blir skrevet inn, skal programmet skrive ut en setning som sier "I am more [adjektiv]!". Programmet skal fortsette å be om adjektiver og skrive ut setninger inntil den totale lengden av alle adjektivene som er skrevet inn overstiger 42 tegn. Når dette skjer, skal programmet avsluttes.

Oppgave 4. 5 poeng

Skriv en funksjon som gjør følgende:

1. Skriver ut 5-gangen mellom 20 og 81 (inkludert) på en linje.
2. Skriver ut tallene fra 48 til 80 (inkludert) med en økning på 8 på en ny linje.
3. Skriver ut tallene fra 100 til 80 (inkludert) med en reduksjon på 3 på en tredje linje.

Funksjonen skal ikke ta inn noen argumenter og skal ikke returnere noe. Den skal kun skrive ut tallene som beskrevet ovenfor.

Eksempel på kjøring:

```
print_numbers()
>>> 5-multiplication table between 20 and 81:
20 25 30 35 40 45 50 55 60 65 70 75 80
Numbers from 48 to 80 incrementing by 8:
48 56 64 72 80
Numbers backwards from 100 to 80 incrementing by -3:
100 97 94 91 88 85 82
```

Oppgave 5. 5 poeng

Skriv en funksjon som skriver ut hvert element i en liste med tall fra 1 til 5 ved hjelp av en for-løkke og enumerate-funksjonen. Funksjonen skal ikke ta inn noen parametere og skal ikke returnere noe.

Eksempel på kjøring:

```
print_numbers_enumerate()
>>> 1
2
3
4
5
```

Oppgave 6. 5 poeng

Skriv en funksjon som skriver ut tallene fra 1 til 15 i reversert rekkefølge ved hjelp av en for-løkke og innebygde Python-funksjoner.

Eksempel på kjøring:

```
print_numbers_for_reversed()
>>> 15
14
13
12
11
10
..
4
3
2
1
```

Oppgave 7. 5 poeng

Skriv en funksjon som tar inn 7 heltall fra brukeren ved hjelp av en while-løkke, og deretter beregner og skriver ut summen av disse tallene.

Oppgave 8. 5 poeng

Skriv en funksjon som multipliserer sammen tallene fra 1 til 1000, men stopper multiplikasjonen når produktet blir større enn 1000. Funksjonen skal returnere det siste produktet som er mindre enn eller lik 1000.

Eksempel på kjøring:

```
print(multiply_for_range())
>>> 5040
```

Oppgave 9. 5 poeng

Skriv et program som spør brukeren om hovedstaden i Niue. Programmet skal fortsette å spørre brukeren inntil riktig svar er gitt. Når brukeren gir riktig svar, skal programmet skrive ut en melding som bekrefter at svaret er riktig, samt antall forsøk brukeren har brukt for å komme frem til riktig svar. Programmet skal ikke være case-sensitive, det vil si at det skal akseptere svar uavhengig av store og små bokstaver.

Oppgave 10. 5 poeng

Kombinasjoner av studenter og fag

Du skal lage et program som tar inn antall studenter og antall fag. Programmet skal deretter generere og skrive ut alle mulige kombinasjoner av studenter og fag, der hver student elsker hvert fag. Du kan anta at antall studenter og fag er positive heltall.

Oppgave 11. 5 poeng

Skriv en funksjon som genererer og skriver ut alle mulige tidspunkt i et døgn med formatet "HH:mm", der HH representerer timer (00-23) og mm representerer minutter (00-59).

Eksempel på kjøring:

```
print_timestamps_v()
>>> 00:00
00:01
00:02
00:03
00:04
00:05
..
23:56
23:57
```

```
23:58
23:59
```

Oppgave 12. 5 poeng

Skriv en funksjon som genererer og skriver ut multiplikasjonstabellen for tallene fra 1 til 10. Hver linje skal vise et multiplikasjonsuttrykk og resultatet, og det skal være en tom linje mellom hver tabell.

Eksempel på kjøring:

```
multiplication_tables()
>>> 1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
1 * 5 = 5
1 * 6 = 6
..
10 * 9 = 90
10 * 10 = 100
```

Oppgave 13. 5 poeng

Hva er output av følgende kjøring?

```
def floor_divide(n=345):
    result = ""
    while n > 0:
        n //= 2
        result += str(n) + " "
    return result.strip()

print(floor_divide(345))
```

Oppgave 14. 5 poeng

Hva er output av følgende kjøring?

```
def print_even_numbers_v():
    for i in range(11):
        if i % 2 == 0:
            if i % 4 == 0:
                print(i, "divisible by 4")
            else:
```

```
        print(i)

print_even_numbers_v()
```

Oppgave 15. 5 poeng

Hva er output av følgende kjøring?

```
def multiply_with_while_loop():
    result = 1
    count = 0
    while count < 10:
        result *= 2
        count += 1
    print(result)

multiply_with_while_loop()
```

Oppgave 16. 5 poeng

Hva er output av følgende kjøring?

```
def multiply_with_list_comprehension():
    value = 1
    value = [value * 2 for _ in range(3)][-1]
    print(value)

multiply_with_list_comprehension()
```

Oppgave 17. 5 poeng

Hva er output av følgende kjøring?

```
def print_stars_recursion(i=5, x=0):
    if x < i:
        print('*' * (x + 1))
        print_stars_recursion(i, x + 1)

print_stars_recursion(5, 0)
```

Oppgave 18. 5 poeng

Lag et program som lar brukeren gjette et tilfeldig tall mellom et gitt nedre og øvre grense. Programmet skal gi tilbakemelding om brukerens gjett er for høyt eller for lavt, og fortsette å be om gjett til brukeren gjetter riktig tall. Bruk en rekursiv funksjon for å implementere dette. Funksjonen skal ta inn nedre og øvre grense som argumenter og generere et tilfeldig tall mellom disse grensene.

Oppgave 19. 5 poeng

Skriv en funksjon som beregner summen av en geometrisk rekke. Funksjonen skal ta inn to parametere: antall ledd 'n' og felles forhold 'r' (med en standardverdi på 2). Funksjonen skal returnere den totale summen av rekken.

Eksempel på kjøring:

```
print(geometric_sum(35, 2))
>>> 68719476735
```

Oppgave 20. 5 poeng

Skriv en funksjon som beregner det n-te Fibonacci-tallet ved hjelp av memoisering. Funksjonen skal ta inn et heltall n som argument og returnere det n-te Fibonacci-tallet. Memoisering skal brukes for å forbedre ytelsen og unngå unødvendige beregninger.

Eksempel på kjøring:

```
print(fibonacci_memoization(24, {}))
>>> 46368
```

Oppgave 21. 5 poeng

Skriv en funksjon som tar inn et heltall 'n' og returnerer summen av de første 'n' Fibonacci-tallene. Hvis 'n' er mindre enn eller lik 0, skal funksjonen returnere 0. Fibonacci-sekvensen starter med tallene 0 og 1, og hvert påfølgende tall er summen av de to foregående tallene (0, 1, 1, 2, 3, 5, 8, ...).

Eksempel på kjøring:

```
print(fibonacci_sum(8))
>>> 33
```

Oppgave 22. 5 poeng

Skriv en funksjon som tar inn et heltall 'n' og returnerer en liste med de første 'n' Fibonacci-tallene ved hjelp av rekursjon. Hvis 'n' er mindre eller lik 0, skal funksjonen returnere en tom liste.

Eksempel på kjøring:

```
print(fibonacci_recursive(20))
>>> [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181]
```

Oppgave 23. 5 poeng

Skriv en funksjon som tar inn et heltall n og returnerer summen av en tallserie der hvert oddetall i serien er kvadrert og lagt til totalen, mens hvert partall i serien er kvadrert og trukket fra totalen. Serien skal starte fra 1 og gå opp til og med n.

Eksempel på kjøring:

```
print(sum_of_series(14))
>>> -105
```

Oppgave 24. 5 poeng

Lag et enkelt Hangman-spill i Python. Spillet skal ta inn to parametere: et ord som skal gjettes og et maksimalt antall forsøk. Spillet skal be brukeren om å gjette en bokstav om gangen og vise gjeldende status for det gjette ordet med understreker for ikke-gjettede bokstaver. Hvis brukeren gjetter riktig bokstav, skal den vises på riktig plass i ordet. Hvis brukeren gjetter feil, skal antall forsøk reduseres med \xe9n. Spillet fortsetter til enten brukeren har gjettet ordet eller brukt opp alle forsøkene. Til slutt skal spillet vise en melding om brukeren vant eller tapte, og avsløre det riktige ordet.

Demo 26.05 1

NTNU
Kunnskap for en bedre verden

Poeng: 120

Oppgave 1 5 poeng

Koden under viser en for-løkke som repeterer 3 ganger. For hver runde blir brukeren bedt om å beskrive seg selv med et adjektiv, hvorpå maskinen disser brukeren ved å si at den er snillere, smartere etc.

Kjør koden for å se hvordan den virker. Endre så koden slik at programmet først spør brukeren hvor mange repetisjoner som ønskes, og deretter utfører programmet med ønsket antall repetisjoner av løkka.

```
Hvor mange adjektiv vil du gi? 2
Beskriv deg selv med et adjektiv? snill
Hah, du snill!? Jeg er mye snillere!
Beskriv deg selv med et adjektiv? ond
Hah, du ond!? Jeg er mye ondere!
Takk for nå!
```

```
#endre koden under
for i in range(3):
    adj = input("Beskriv deg selv med et adjektiv? ")
    print("Hah, du", adj + "!? Jeg er mye", adj + "ere!")
print("Takk for nå!")
```

Oppgave 2 5 poeng

Koden under viser en while-løkke som utfører samme type brukerdialog som i (Oppgave 1), med 3 repetisjoner. Kjør den for å se hvordan det virker. ****Endre nå programmet så løkka ikke kjører akkurat 3 ganger, men så mange ganger brukeren vil****, hvor brukeren kan bestemme dette underveis ved å gi et tomt svar (dvs. bare slå Enter) for å slutte. Da vil input()-setningen resultere i en tom streng, "".

```
Slå Enter uten å skrive noe når du vil avslutte.
Beskriv deg selv med et adjektiv? snill
Hah, du snill!? Jeg er mye snillere!
Beskriv deg selv med et adjektiv? smart
Hah, du smart!? Jeg er mye smartere!
Beskriv deg selv med et adjektiv? flittig
```

```
Hah, du flittig!? Jeg er mye flittigere!
Beskriv deg selv med et adjektiv?
Takk for nå!
```

```
while True:
    adj = input("Beskriv deg selv med et adjektiv? ")
    if adj == "":
        break
    print("Hah, du", adj + "!? Jeg er mye", adj + "ere!")
    i += 1  # øker i med 1
print("Takk for nå!")
```

Oppgave 3 5 poeng

Ta igjen utgangspunkt i while-løkka med 3 repetisjoner som vist før (Oppgave 2). Vi ønsker nå å endre programmet på følgende måte:

Brukeren starter med 42 bokstaver til disposisjon. For hver runde skal programmet trekke fra antall bokstaver i det adjektivet som ble brukt.

Løkka skal fortsette så lenge det fortsatt er bokstaver til disposisjon (dvs. dette tallet er større enn 0).

```
Du har 42 bokstaver til disposisjon.
Beskriv deg selv med et adjektiv? snill
Hah, du snill!? Jeg er mye snillere!
Du har 37 bokstaver til disposisjon.
Beskriv deg selv med et adjektiv? desperat
Hah, du desperat!? Jeg er mye desperatere!
Du har 29 bokstaver til disposisjon.
Beskriv deg selv med et adjektiv? kjempetørst
Hah, du kjempetørst!? Jeg er mye kjempetørstere!
Du har 18 bokstaver til disposisjon.
Beskriv deg selv med et adjektiv? megasupereffektiv
Hah, du megasupereffektiv!? Jeg er mye megasupereffektivere!
Du har 1 bokstaver til disposisjon.
Beskriv deg selv med et adjektiv? o
Hah, du o!? Jeg er mye oere!
Takk for nå!
```

I siste linje, hvor det bare er 1 bokstav til disposisjon, er brukeren lojal her og skrive bare en bokstav - men du behøver ikke lage programmet slik at det sikrer dette... det er ok om brukeren skriver et lenger ord den siste gangen, så lenge brukeren ikke får lov til å skrive nye ord når antall bokstaver til disposisjon er blitt <= 0.

Oppgave 4 5 poeng

Koden under er de to første for-løkkene ferdige og gir riktig utskrift i forhold til den forklarende teksten i print-setningene.

De tre neste er ikke ferdige. Bytt ut `range(0)` i disse tre (også merket med ###) slik at for-løkkene gir tallsekvenser som passer med det som forklares i print-setningene.

```
print("Oddetallene fra 1 til 20:")
for number in range(1, 20, 2):
    print(number, end = " ")
print()

print("Tallene i 3-gangen mellom 12 og 25:")
for number in range(12, 25, 3):
    print(number, end = " ")
print()

print("Tallene i 5-gangen mellom 20 og 81:")
for number in range(0): ###
    print(number, end = " ")
print()

print("Tallsekvensen 48, 56, 64, 72, 80")
for number in range(0): ###
    print(number, end = " ")
print()

print("Telle baklengs fra 100 til 80, med intervall på -3, dvs. 100, 97, ...:")
for number in range(0): ###
    print(number, end = " ")
print()
```

Oppgave 5 5 poeng

Lag et program som skriver ut tallene 1 til 5 ved bruk av en for-løkke.

Oppgave 6 5 poeng

Lag et program som teller nedover fra 15 til 1 ved hjelp av en for-løkke. Skriv ut alle tallene.

Oppgave 7 5 poeng

Lag et program som ved hjelp av en løkke ber brukeren om å taste inn 7 heltall, som vist i eksemplet på kjøring under. Til slutt skal programmet skrive ut hva summen av tallene ble.

```
Skriv inn et heltall: 6
Skriv inn et heltall: 4
Skriv inn et heltall: 7
Skriv inn et heltall: 3
Skriv inn et heltall: 2
Skriv inn et heltall: 456
Skriv inn et heltall: 99
Summen av tallene ble 577
```

Oppgave 8 5 poeng

Lag et program som multipliserer sammen alle tallene fra 1,2,3,... og avslutter når produktet er større enn 1000.

Oppgave 9 5 poeng

Lag et program som stiller brukeren det samme spørsmålet, om og om igjen, helt til det korrekte svaret blir skrevet. Da skal programmet fortelle hvor mange forsøk som ble brukt. Eksempel på kjøring vist nedenfor, men du kan godt bytte ut med et spørsmål med annen tematikk, noe du selv er interessert i.

```
Hva heter hovedstaden til Niue? Niue City
Det var feil, prøv igjen.
Hva heter hovedstaden til Niue? Niuania
Det var feil, prøv igjen.
Hva heter hovedstaden til Niue? Apia
Det var feil, prøv igjen.
Hva heter hovedstaden til Niue? Alofi
Korrekt!! Du brukte 4 forsøk.
```

Oppgave 10 5 poeng

Eksempel på nøstet løkke:

```
for x in range(5):
    for y in range(3):
        print("Jeg elsker ITGK! ", end=" ")
    print()
```

Koden over vil skrive ut “Jeg elsker ITGK!” tre ganger ved siden av hverandre fem ganger nedover.

Kopier koden i eksemplet "Jeg elsker ITGK!" over og kjør programmet. Denne utskriften trenger man strengt tatt ikke dobbel løkke for å få til; selve utskriften gir riktig nok et todimensjonalt inntrykk, men siden den underliggende informasjonen ikke er todimensjonal men derimot konstant (samme utskrift "Jeg elsker ITGK!" i alle tilfeller), er det også greit mulig å klare dette med enkel løkke, eller helt uten løkker, som følger:

```
# Løsning med enkel løkke:
for x in range(5):
    print("Jeg elsker ITGK!  Jeg elsker ITGK!  Jeg elsker ITGK!")
```

```
# Løsning uten løkker; "\n" er tegnet for linjeskift (newline på engelsk)
print((("Jeg elsker ITGK!  " * 3) + "\n") * 5)
```

```
Jeg elsker ITGK!  Jeg elsker ITGK!  Jeg elsker ITGK!
Jeg elsker ITGK!  Jeg elsker ITGK!  Jeg elsker ITGK!
Jeg elsker ITGK!  Jeg elsker ITGK!  Jeg elsker ITGK!
Jeg elsker ITGK!  Jeg elsker ITGK!  Jeg elsker ITGK!
Jeg elsker ITGK!  Jeg elsker ITGK!  Jeg elsker ITGK!
```

Hvis vi derimot endrer litt på kravene til programmet, så det skal handle om flere studenter enn "Jeg" og flere emner enn "ITGK", vil vi ha et genuint todimensjonalt problem (én dimensjon er studenter, en annen emner) hvor dobbel løkke vil være klart enkleste løsning. **Din oppgave: endre programmet så det gir utskrift som vist nedenfor**. Fra forrige oppgave skal altså Jeg byttes ut med Stud <nummer>, og ITGK skal byttes ut med Emne <nummer>. Brukeren skal angi ønsket antall for både studenter og emner, 4 og 3 er bare eksempler.

```
Hvor mange studenter? 4
Hvor mange emner? 3
Stud 1 elsker Emne 1 ; Stud 1 elsker Emne 2 ; Stud 1 elsker Emne 3 ;
Stud 2 elsker Emne 1 ; Stud 2 elsker Emne 2 ; Stud 2 elsker Emne 3 ;
Stud 3 elsker Emne 1 ; Stud 3 elsker Emne 2 ; Stud 3 elsker Emne 3 ;
Stud 4 elsker Emne 1 ; Stud 4 elsker Emne 2 ; Stud 4 elsker Emne 3 ;
```

Oppgave 11 5 poeng

Bruk en dobbel løkke til å skrive ut en liste over alle klokkeslett i løpet av et døgn på en fin måte. Du trenger kun å tenke på timer og minutter. Klokkeslettene skal gå fra 0:0 til 23:59 som vist nedenfor.

Eksempel på kjøring av kode:

```
0:0
0:1
0:2
.
.	# Alle klokkeslett i mellom her skal skrives ut
.
23:58
23:59
```

Oppgave 12 5 poeng

Bruk doble løkker til å skrive ut alle tallene i den lille gangetabellen, dvs. lag to for-løkker opp til 11 og multipliser variabelen x fra den ene løkken med variabelen y fra den andre løkken og skriv det ut i den innerste løkken. Du trenger ikke å tenke på formatet til utskriften.

Eksempel på kjøring:

```
1
2
3
4
5
.
.
.
60
70
80
90
100
```

Oppgave 13 5 poeng

Hva skrives ut i koden under?

```
a=345
b="
while a or b==":
```

```
    b=str(a%2)+b
    a=a//2
print(b)
```

Oppgave 14 5 poeng

Hva skrives ut i koden under?

```
for x in range(0, 10, 2):
    print(x, end=")
    if x%4==0:
        print( ": Dette tallet går opp i 4-gangern")
    else:
        print()
```

`end=""` gjør at det neste som printes ikke printes en linje under, men at det fortsetter på samme linje.

Oppgave 15 5 poeng

Hva skrives ut i koden under?

```
i = 1
while i<10:
    i = i*2
print(i)
```

Oppgave 16 5 poeng

Hva skrives ut i koden under?

```
i = 1
j = 3
while j>0:
    i = i*2
    j = j - 1
print(i)
```

Oppgave 17 5 poeng

Hva skrives ut i koden under?

```
i = 5
for x in range(i):
    for y in range(x+1):
        print("*", end="")
```

```
    print()
```

Oppgave 18 5 poeng

I denne oppgaven skal du lage et program som genererer et tilfeldig heltall i et gitt intervall, og deretter lar brukeren gjette på hvilket tall dette er. Dette bør gjøres ved bruk av løkker.

1. Be brukeren om å velge en nedre og en øvre grense for tall han eller hun skal gjette på. Lagre disse to opplysningene i to variabler.

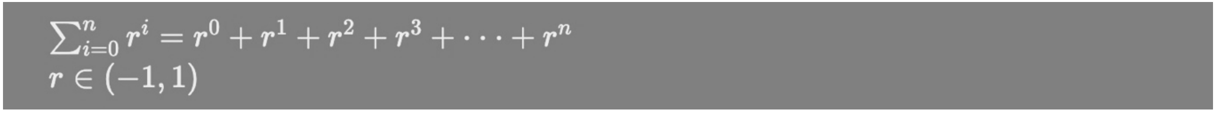
2. Lag en variabel TilfeldigTall som genererer et tilfeldig tall i intervallet mellom den øvre og den nedre grensen som brukeren har satt.

3. Skriv en while-løkke som kjører så lenge brukeren ikke har gjettet riktig svar. Brukeren skal få tilbakemelding for hvert gjett om han eller hun gjettet for lavt, for høyt eller riktig.

```
Gi en nedre grense for det tilfeldige tallet: 1
Gi en øvre grense for det tilfeldige tallet: 100
Make a guess 50
The correct number is lower
Make a guess 25
The correct Number is higher
Make a guess 37
The correct number is lower
Make a guess 32
You guessed correct!
```

Oppgave 19 5 poeng

En geometrisk rekke er en sum som kan skrives på formen under:



$$\sum_{i=0}^{n} r^i = r^0 + r^1 + r^2 + r^3 + \cdots + r^n$$
$$r \in (-1, 1)$$

Lag et program som summerer en geometrisk rekke fra 0 til n ved hjelp av en while løkke.

Sjekk: r = 0.5 og n = 4 skal gi sum = 1.9375

Oppgave 20 5 poeng

Fibonaccitallene er definert som følger:

- f(0)=0
- f(1)=1
- f(k)=f(k−1)+f(k−2)

Det vil si at de to første tallene i rekken er 0 og 1, deretter er det neste tallet summen av de to forrige tallene. Starten på rekken ser derfor slik ut: 0 1 1 2 3 5 8 13 ...

Lag et program som regner ut og returnerer det k-te fibonaccitallet f(k) ved hjelp av iterasjon. Har du gjort det rett skal det 10-ende Fibonaccitallet bli 34. **Husk at det første tallet i rekken er tall nummer 0**

Oppgave 21 5 poeng

Skriv om programmet i (Oppgave 20) slik at det også regner ut summen av alle fibonaccitallene. Har du gjort det rett det rett skal summen av Fibonaccitallene opp til 10 bli 88.

Oppgave 22 5 poeng

Modifiser programmet i (Oppgave 20) til å returnere en liste med alle fibonaccitallene opp til og med f(k).

Husk å skrive ut svarene til skjerm.

Oppgave 23 5 poeng

Skriv et program som leser inn et heltall n fra bruker og legger sammen tallserien under.

$$1^2 - 2^2 + 3^2 - 4^2 + 5^2 - \cdots \pm n^2$$

Legg merke til at alle partallene har negativt fortegn og alle oddetallene har positivt fortegn. Husk at navnet på variabelen din *ikke* kan være **sum**, ettersom dette er navnet på en funksjon i python. Husk også at range() bare går til et tall og ikke til og med.

```
n = 7
Summen av tallserien er 28
```

Oppgave 24 5 poeng

I denne oppgaven bruker vi en enkel while-løkke for å lage et hangman-spill i Python.

Lag et program som tar inn et ord (tekststreng) og antall liv (heltall) fra en bruker, og lar en annen (eller samme) bruker gjette på bokstaver i ordet.

1. Start med å hente inn data fra bruker. Lagre dette i to variabler "hemmelig_ord" og "antall_liv".

2. Under har vi laget en while-løkke som kjører evig. Din oppgave er å fylle inn manglende logikk inne i løkken. Ting som må gjøres er:
- Hent inn en bokstav fra bruker
- Sjekk om denne er i det hemmelige ordet
- Trekk fra et liv dersom brukeren tipper feil
- Hvis brukeren ikke har flere liv skal løkken avsluttes

PS: Husk å skrive ut resultatet til brukeren. **Du kan bruke variablene du laget i oppgave a uten å skrive dem på nytt**

```
Skriv inn det hemmelige ordet: hemmelig
Hvor mange forsøk får brukeren? 2
Gjett på én bokstav i ordet: f
Bokstaven f er ikke i ordet.
Du har  1 liv igjen, prøv på nytt.
Gjett på én bokstav i ordet: h
Stemmer, bokstaven er i ordet
Gjett på én bokstav i ordet: e
Stemmer, bokstaven er i ordet
Gjett på én bokstav i ordet: r
Bokstaven r er ikke i ordet.
Du har ingen liv igjen.
```

Løsningsforslag Demo 26.05 1

NTNU
Kunnskap for en bedre verden

Generert: 26 May, 2023 18:58:10

Poeng: 120

Oppgave 1. 5 poeng

```
def more_adjective():
    adjective = input("Enter an adjective: ")
    range_num = int(input("Enter a range: "))
    i = 1
    while i <= range_num:
        print(f"I am {i} times more {adjective}!")
        i += 1
```

Oppgave 2. 5 poeng

```
def more_adjective():
    adjective = input("Enter an adjective: ")
    while adjective:
        print(f"I am more {adjective}!")
        adjective = input("Enter another adjective: ")
```

Oppgave 3. 5 poeng

```
def more_adjective_for_range():
    total_chars = 0
    for _ in range(42):
        if total_chars >= 42:
            break
        adjective = input("Enter an adjective: ")
        total_chars += len(adjective)
        print(f"I am more {adjective}!")
```

Oppgave 4. 5 poeng

```
def print_numbers():
    print("5-multiplication table between 20 and 81:")
    print(*(i for i in range(20, 82, 5)))
    print("Numbers from 48 to 80 incrementing by 8:")
```

```
    print(*(i for i in range(48, 81, 8)))
    print("Numbers backwards from 100 to 80 incrementing by -3:")
    print(*(i for i in range(100, 79, -3)))
```

Oppgave 5. 5 poeng

```
def print_numbers():
    i = 1
    while i <= 5:
        print(i)
        i += 1
```

Oppgave 6. 5 poeng

```
def print_numbers_recursion(i=15):
    if i > 0:
        print(i)
        print_numbers_recursion(i - 1)
```

Oppgave 7. 5 poeng

```
def sum_numbers_list_comprehension():
    numbers = [int(input("Enter a number: ")) for _ in range(7)]
    total = sum(numbers)
    print("The sum is:", total)
```

Oppgave 8. 5 poeng

```
def multiply_recursive(product=1, i=1):
    if product > 1000:
        return product
    return multiply_recursive(product * i, i + 1)
```

Oppgave 9. 5 poeng

```
def capital_niue():
    capital = "Alofi"
    tries = 0
    while True:
        user_input = input("What is the capital of Niue? ")
        tries += 1
        if user_input.lower() == capital.lower():
            print(f"Correct! The capital of Niue is {capital}.")
```

```
            print(f"It took you {tries} tries.")
            break
        else:
            print("Incorrect. Please try again.")
```

Oppgave 10. 5 poeng

```
def student_subject_combinations():
    students = int(input("Enter the number of students: "))
    subjects = int(input("Enter the number of subjects: "))
    student = 1
    while student <= students:
        for subject in range(1, subjects + 1):
            print(f"Student {student} loves subject {subject}")
        student += 1
```

Oppgave 11. 5 poeng

```
def print_timestamps_v():
    hour = 0
    while hour < 24:
        for minute in range(60):
            print(f"{hour:02d}:{minute:02d}")
        hour += 1
```

Oppgave 12. 5 poeng

```
def multiplication_tables():
    i = 1
    while i <= 10:
        for j in range(1, 11):
            print(f"{i} * {j} = {i * j}")
        print("\n")
        i += 1
```

Oppgave 13. 5 poeng

```
>>> 172 86 43 21 10 5 2 1 0
```

Oppgave 14. 5 poeng

```
>>> 0 divisible by 4
2
```

```
4 divisible by 4
6
8 divisible by 4
10
```

Oppgave 15. 5 poeng

```
>>> 1024
```

Oppgave 16. 5 poeng

```
>>> 8
```

Oppgave 17. 5 poeng

```
>>> *
**
***
****
*****
```

Oppgave 18. 5 poeng

```
def guess_number_while_try_except():
    lower_limit = int(input("Enter the lower limit: "))
    upper_limit = int(input("Enter the upper limit: "))
    random_number = random.randint(lower_limit, upper_limit)
    print("Guess the number between", lower_limit, "and", upper_limit)
    while True:
        try:
            guess = int(input("Enter your guess: "))
            if guess < random_number:
                print("Go higher!")
            elif guess > random_number:
                print("Go lower!")
            else:
                print("Congratulations! You guessed the number!")
                break
        except ValueError:
            print("Invalid input. Please enter a number.")
```

Oppgave 19. 5 poeng

```
def geometric_sum_recursive(n, r=2):
    if n == 0:
        return 1
    else:
        return r ** n + geometric_sum_recursive(n - 1, r)
```

Oppgave 20. 5 poeng

```
def fibonacci_recursive(n):
    if n <= 1:
        return n
    else:
        return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)
```

Oppgave 21. 5 poeng

```
def fibonacci_generator(n):
    a, b = 0, 1
    for _ in range(n):
        yield a
        a, b = b, a + b
```

Oppgave 22. 5 poeng

```
def fibonacci_loop(n):
    fib_list = []
    a, b = 0, 1
    while a <= n:
        fib_list.append(a)
        a, b = b, a + b
    return fib_list
```

Oppgave 23. 5 poeng

```
def sum_of_series(n):
    return sum([(-1) ** (i % 2) * i ** 2 for i in range(1, n + 1)])
```

Oppgave 24. 5 poeng

```
def hangman_v(word, tries):
    guessed_word = ['_' for _ in word]
    attempts = 0
    while attempts < tries:
```

```
        guess = input("Guess a letter: ").lower()
        if guess in word:
            guessed_word = [letter if letter == guess else char for char, letter in zip(guessed_word, word)]
        else:
            attempts += 1
        print(" ".join(guessed_word))
        if "_" not in guessed_word:
            print("Congratulations! You won!")
            break
    else:
        print(f"Sorry, you lost. The word was '{word}'.")
```

Løsningsforslag Demo 26.05 2

NTNU
Kunnskap for en bedre verden

Generert: 26 May, 2023 18:58:12

Poeng: 120

Oppgave 1. 5 poeng

```
def more_adjective_list_comprehension():
    adjective = input("Enter an adjective: ")
    range_num = int(input("Enter a range: "))
    sentences = [print(f"I am {i + 1} times more {adjective}!") for i in range(range_num)]
```

Oppgave 2. 5 poeng

```
def more_adjective():
    adjective = input("Enter an adjective: ")
    while adjective:
        print(f"I am more {adjective}!")
        adjective = input("Enter another adjective: ")
```

Oppgave 3. 5 poeng

```
def more_adjective():
    total_chars = 0
    while total_chars < 42:
        adjective = input("Enter an adjective: ")
        total_chars += len(adjective)
        print(f"I am more {adjective}!")
```

Oppgave 4. 5 poeng

```
def print_numbers():
    print("5-multiplication table between 20 and 81:")
    print(*[i for i in range(20, 82, 5)])
    print("Numbers from 48 to 80 incrementing by 8:")
    print(*[i for i in range(48, 81, 8)])
    print("Numbers backwards from 100 to 80 incrementing by -3:")
    print(*[i for i in range(100, 79, -3)])
```

Oppgave 5. 5 poeng

```
def print_numbers():
    for i in range(1, 6):
        print(i)
```

Oppgave 6. 5 poeng

```
def print_numbers_for_range():
    for i in range(15, 0, -1):
        print(i)
```

Oppgave 7. 5 poeng

```
def sum_numbers_for_loop():
    total = 0
    for i in range(7):
        number = int(input("Enter a number: "))
        total += number
    print("The sum is:", total)
```

Oppgave 8. 5 poeng

```
def multiply_for_iterator():
    product = 1
    iterator = iter(range(1, 1000))
    for i in iterator:
        product *= i
        if product > 1000:
            break
    return product
```

Oppgave 9. 5 poeng

```
def capital_niue():
    capital = "Alofi"
    tries = 0
    while True:
        user_input = input("What is the capital of Niue? ")
        tries += 1
        if user_input.lower() == capital.lower():
            print(f"Correct! The capital of Niue is {capital}.")
            print(f"It took you {tries} tries.")
            break
```

```
    else:
        print("Incorrect. Please try again.")
```

Oppgave 10. 5 poeng

```
def student_subject_combinations():
    students = int(input("Enter the number of students: "))
    subjects = int(input("Enter the number of subjects: "))
    student = 1
    while student <= students:
        subject = 1
        while subject <= subjects:
            print(f"Student {student} loves subject {subject}")
            subject += 1
        student += 1
```

Oppgave 11. 5 poeng

```
def print_timestamps_v():
    for time in range(24 * 60):
        hour = time // 60
        minute = time % 60
        print(f"{hour:02d}:{minute:02d}")
```

Oppgave 12. 5 poeng

```
def multiplication_tables():
    for i in range(1, 11):
        table = [f"{i} * {j} = {i * j}" for j in range(1, 11)]
        for row in table:
            print(row)
        print("\n")
```

Oppgave 13. 5 poeng

```
>>> 172 86 43 21 10 5 2 1 0
```

Oppgave 14. 5 poeng

```
>>> 0 divisible by 4
2
4 divisible by 4
6
```

```
8 divisible by 4
10
```

Oppgave 15. 5 poeng

```
>>> 1024
```

Oppgave 16. 5 poeng

```
>>> 8
```

Oppgave 17. 5 poeng

```
>>> *
**
***
****
*****
```

Oppgave 18. 5 poeng

```
def guess_number():
    lower_limit = int(input("Enter the lower limit: "))
    upper_limit = int(input("Enter the upper limit: "))
    random_number = random.randint(lower_limit, upper_limit)
    print("Guess the number between", lower_limit, "and", upper_limit)
    for _ in range(1000000):
        guess = int(input("Enter your guess: "))
        if guess < random_number:
            print("Go higher!")
        elif guess > random_number:
            print("Go lower!")
        else:
            print("Congratulations! You guessed the number!")
            break
```

Oppgave 19. 5 poeng

```
def geometric_sum_list_comprehension(n, r=2):
    return sum([r ** i for i in range(n + 1)])
```

Oppgave 20. 5 poeng

```
def fibonacci_iterative(n):
    if n <= 1:
        return n
    a, b = 0, 1
    for _ in range(2, n + 1):
        a, b = b, a + b
    return b
```

Oppgave 21. 5 poeng

```
def fibonacci_sum_recursive(n):
    if n <= 0:
        return 0
    if n == 1:
        return 1
    return fibonacci_sum_recursive(n - 1) + fibonacci_sum_recursive(n - 2) + 1
```

Oppgave 22. 5 poeng

```
def fibonacci_memo(n, memo={}):
    if n in memo:
        return memo[n]
    if n <= 0:
        return []
    if n == 1:
        return [0]
    if n == 2:
        return [0, 1]
    fib_list = fibonacci_memo(n - 1)
    fib_list.append(fib_list[-1] + fib_list[-2])
    memo[n] = fib_list
    return fib_list
```

Oppgave 23. 5 poeng

```
def sum_of_series(n):
    total = 0
    for i in range(1, n + 1):
        if i % 2 == 0:
            total -= i ** 2
        else:
            total += i ** 2
    return total
```

Oppgave 24. 5 poeng

```
def hangman_v(word, tries):
    guessed_word = ['_' for _ in word]
    attempts = 0
    while attempts < tries:
        guess = input("Guess a letter: ").lower()
        if guess in word:
            for i, letter in enumerate(word):
                if letter == guess:
                    guessed_word[i] = guess
        else:
            attempts += 1
        print(" ".join(guessed_word))
        if set(guessed_word) == set(word):
            print("Congratulations! You won!")
            break
    else:
        print(f"Sorry, you lost. The word was '{word}'.")
```

Løsningsforslag Demo 26.05 3

NTNU
Kunnskap for en bedre verden

Generert: 26 May, 2023 18:58:14

Poeng: 120

Oppgave 1. 5 poeng

```
def more_adjective_recursive(adjective=None, range_num=None, i=1):
    if adjective is None and range_num is None:
        adjective = input("Enter an adjective: ")
        range_num = int(input("Enter a range: "))
    if i <= range_num:
        print(f"I am {i} times more {adjective}!")
        more_adjective_recursive(adjective, range_num, i + 1)
```

Oppgave 2. 5 poeng

```
def more_adjective_recursive():
    adjective = input("Enter an adjective: ")
    if not adjective:
        return
    print(f"I am more {adjective}!")
    more_adjective_recursive()
```

Oppgave 3. 5 poeng

```
def more_adjective_while_try_except():
    total_chars = 0
    while True:
        try:
            if total_chars >= 42:
                break
            adjective = input("Enter an adjective: ")
            total_chars += len(adjective)
            print(f"I am more {adjective}!")
        except:
            continue
```

Oppgave 4. 5 poeng

```
def print_numbers():
    print("5-multiplication table between 20 and 81:")
    for i in range(20, 82, 5):
        print(i, end=" ")
    print("\nNumbers from 48 to 80 incrementing by 8:")
    for i in range(48, 81, 8):
        print(i, end=" ")
    print("\nNumbers backwards from 100 to 80 incrementing by -3:")
    for i in range(100, 79, -3):
        print(i, end=" ")
```

Oppgave 5. 5 poeng

```
def print_numbers_list_comprehension():
    [print(i) for i in range(1, 6)]
```

Oppgave 6. 5 poeng

```
def print_numbers():
    i = 15
    while i > 0:
        print(i)
        i -= 1
```

Oppgave 7. 5 poeng

```
def sum_numbers_map():
    numbers = []
    for _ in range(7):
        number = int(input("Enter a number: "))
        numbers.append(number)
    total = sum(map(int, numbers))
    print("The sum is:", total)
```

Oppgave 8. 5 poeng

```
def multiply():
    product = 1
    i = 1
    while product <= 1000:
        product *= i
        i += 1
    return product
```

Oppgave 9. 5 poeng

```
def capital_niue():
    capital = "Alofi"
    tries = 0
    for _ in range(10000):
        user_input = input("What is the capital of Niue? ")
        tries += 1
        if user_input.lower() == capital.lower():
            print(f"Correct! The capital of Niue is {capital}.")
            print(f"It took you {tries} tries.")
            break
        else:
            print("Incorrect. Please try again.")
```

Oppgave 10. 5 poeng

```
def student_subject_combinations():
    students = int(input("Enter the number of students: "))
    subjects = int(input("Enter the number of subjects: "))
    for student in range(1, students + 1):
        for subject in range(1, subjects + 1):
            print(f"Student {student} loves subject {subject}")
```

Oppgave 11. 5 poeng

```
def print_timestamps_v():
    for hour in range(24):
        for minute in range(60):
            print(f"{hour:02d}:{minute:02d}")
```

Oppgave 12. 5 poeng

```
def multiplication_tables():
    i = 1
    while i <= 10:
        j = 1
        while j <= 10:
            print(f"{i} * {j} = {i * j}")
            j += 1
        print("\n")
        i += 1
```

Oppgave 13. 5 poeng

```
>>> 172 86 43 21 10 5 2 1 0
```

Oppgave 14. 5 poeng

```
>>> 0 divisible by 4
2
4 divisible by 4
6
8 divisible by 4
10
```

Oppgave 15. 5 poeng

```
>>> 1024
```

Oppgave 16. 5 poeng

```
>>> 8
```

Oppgave 17. 5 poeng

```
>>> *
**
***
****
*****
```

Oppgave 18. 5 poeng

```
def guess_number():
    lower_limit = int(input("Enter the lower limit: "))
    upper_limit = int(input("Enter the upper limit: "))
    random_number = random.randint(lower_limit, upper_limit)
    print("Guess the number between", lower_limit, "and", upper_limit)
    while True:
        guess = int(input("Enter your guess: "))
        if guess < random_number:
            print("Go higher!")
        elif guess > random_number:
            print("Go lower!")
        else:
```

```
        print("Congratulations! You guessed the number!")
        break
```

Oppgave 19. 5 poeng

```
def geometric_sum(n, r=2):
    total = 0
    i = 0
    while i <= n:
        total += r ** i
        i += 1
    return total
```

Oppgave 20. 5 poeng

```
def fibonacci_recursive(n):
    if n <= 1:
        return n
    else:
        return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)
```

Oppgave 21. 5 poeng

```
def fibonacci_sum(n):
    if n <= 0:
        return 0
    a, b = 0, 1
    sum_fib = 0
    count = 0
    while count < n:
        sum_fib += a
        a, b = b, a + b
        count += 1
    return sum_fib
```

Oppgave 22. 5 poeng

```
def fibonacci_loop(n):
    fib_list = []
    a, b = 0, 1
    while a <= n:
        fib_list.append(a)
        a, b = b, a + b
    return fib_list
```

Oppgave 23. 5 poeng

```
def sum_of_series(n):
    return sum(((-1) ** (i % 2) * i ** 2 for i in range(1, n + 1)))
```

Oppgave 24. 5 poeng

```
def hangman_v(word, tries):
    guessed_word = ['_' for _ in word]
    attempts = 0
    while attempts < tries:
        guess = input("Guess a letter: ").lower()
        if guess in word:
            guessed_word = [letter if letter == guess else guessed_word[i] for i, letter in enumerate(word)]
        else:
            attempts += 1
        print(" ".join(guessed_word))
        if "_" not in guessed_word:
            print("Congratulations! You won!")
            break
    else:
        print(f"Sorry, you lost. The word was '{word}'.")
```

Løsningsforslag Demo 26.05 4

NTNU
Kunnskap for en bedre verden

Generert: 26 May, 2023 18:58:16

Poeng: 120

Oppgave 1. 5 poeng

```
def more_adjective():
    adjective = input("Enter an adjective: ")
    range_num = int(input("Enter a range: "))
    for i in range(range_num):
        print(f"I am {i + 1} times more {adjective}!")
```

Oppgave 2. 5 poeng

```
def more_adjective_for_break():
    while True:
        adjective = input("Enter an adjective: ")
        if not adjective:
            break
        print(f"I am more {adjective}!")
```

Oppgave 3. 5 poeng

```
def more_adjective_for_count():
    total_chars = 0
    for _ in itertools.count():
        if total_chars >= 42:
            break
        adjective = input("Enter an adjective: ")
        total_chars += len(adjective)
        print(f"I am more {adjective}!")
```

Oppgave 4. 5 poeng

```
def print_numbers():
    print("5-multiplication table between 20 and 81:")
    i = 20
    while i <= 81:
        print(i, end=" ")
```

```
        i += 5
    print("\nNumbers from 48 to 80 incrementing by 8:")
    i = 48
    while i <= 80:
        print(i, end=" ")
        i += 8
    print("\nNumbers backwards from 100 to 80 incrementing by -3:")
    i = 100
    while i >= 80:
        print(i, end=" ")
        i -= 3
```

Oppgave 5. 5 poeng

```
def print_numbers_enumerate():
    numbers = [1, 2, 3, 4, 5]
    for i, num in enumerate(numbers, start=1):
        print(num)
```

Oppgave 6. 5 poeng

```
def print_numbers_for_reversed():
    for i in reversed(range(1, 16)):
        print(i)
```

Oppgave 7. 5 poeng

```
def sum_numbers_while_loop():
    total = 0
    count = 0
    while count < 7:
        number = int(input("Enter a number: "))
        total += number
        count += 1
    print("The sum is:", total)
```

Oppgave 8. 5 poeng

```
def multiply_for_range():
    product = 1
    for i in range(1, 1000):
        product *= i
        if product > 1000:
            break
```

```
    return product
```

Oppgave 9. 5 poeng

```
def capital_niue_while_counter():
    capital = "Alofi"
    tries = 0
    correct = False
    while not correct:
        user_input = input("What is the capital of Niue? ")
        tries += 1
        if user_input.lower() == capital.lower():
            correct = True
            print(f"Correct! The capital of Niue is {capital}.")
            print(f"It took you {tries} tries.")
        else:
            print("Incorrect. Please try again.")
```

Oppgave 10. 5 poeng

```
def student_subject_combinations():
    students = int(input("Enter the number of students: "))
    subjects = int(input("Enter the number of subjects: "))
    for student, subject in itertools.product(range(1, students + 1), range(1, subjects + 1)):
        print(f"Student {student} loves subject {subject}")
```

Oppgave 11. 5 poeng

```
def print_timestamps_v():
    for hour in range(24):
        for minute in range(60):
            print(f"{hour:02d}:{minute:02d}")
```

Oppgave 12. 5 poeng

```
def multiplication_tables():
    for i in range(1, 11):
        for j in range(1, 11):
            print(f"{i} * {j} = {i * j}")
        print("\n")
```

Oppgave 13. 5 poeng

```
>>> 172 86 43 21 10 5 2 1 0
```

Oppgave 14. 5 poeng

```
>>> 0 divisible by 4
2
4 divisible by 4
6
8 divisible by 4
10
```

Oppgave 15. 5 poeng

```
>>> 1024
```

Oppgave 16. 5 poeng

```
>>> 2
```

Oppgave 17. 5 poeng

```
>>> *
**
***
****
*****
```

Oppgave 18. 5 poeng

```
def guess_number_recursive(lower_limit, upper_limit, random_number=None):
    if random_number is None:
        lower_limit = int(input("Enter the lower limit: "))
        upper_limit = int(input("Enter the upper limit: "))
        random_number = random.randint(lower_limit, upper_limit)
        print("Guess the number between", lower_limit, "and", upper_limit)
    guess = int(input("Enter your guess: "))
    if guess < random_number:
        print("Go higher!")
        guess_number_recursive(lower_limit, upper_limit, random_number)
    elif guess > random_number:
        print("Go lower!")
        guess_number_recursive(lower_limit, upper_limit, random_number)
    else:
```

```
        print("Congratulations! You guessed the number!")
```

Oppgave 19. 5 poeng

```
def geometric_sum(n, r=2):
    total = 0
    for i in range(n + 1):
        total += r ** i
    return total
```

Oppgave 20. 5 poeng

```
def fibonacci_memoization(n, memo={}):
    if n <= 1:
        return n
    elif n not in memo:
        memo[n] = fibonacci_memoization(n - 1) + fibonacci_memoization(n - 2)
    return memo[n]
```

Oppgave 21. 5 poeng

```
def fibonacci_sum(n):
    if n <= 0:
        return 0
    a, b = 0, 1
    sum_fib = 0
    for _ in range(n):
        sum_fib += a
        a, b = b, a + b
    return sum_fib
```

Oppgave 22. 5 poeng

```
def fibonacci_recursive(n):
    if n <= 0:
        return []
    if n == 1:
        return [0]
    if n == 2:
        return [0, 1]
    fib_list = fibonacci_recursive(n - 1)
    fib_list.append(fib_list[-1] + fib_list[-2])
    return fib_list
```

Oppgave 23. 5 poeng

```
def sum_of_series(n):
    total = 0
    for i in range(1, n + 1):
        if i % 2 == 0:
            total -= i ** 2
        else:
            total += i ** 2
    return total
```

Oppgave 24. 5 poeng

```
def hangman_v(word, tries):
    guessed_word = ['_' for _ in word]
    attempts = 0
    while attempts < tries:
        guess = input("Guess a letter: ").lower()
        if guess in word:
            for i, letter in enumerate(word):
                if letter == guess:
                    guessed_word[i] = guess
        else:
            attempts += 1
        print(" ".join(guessed_word))
        if "_" not in guessed_word:
            print("Congratulations! You won!")
            break
    else:
        print(f"Sorry, you lost. The word was '{word}'.")
```

NTNU

Norwegian University of
Science and Technology