Generating Test data for Table driven Tests with different LLMs to evaluate their potential for test automation

1st xnacly
Applied computer science
DHBW

2nd hlxid
Applied computer science
DHBW

***Abstract*—This paper aims to reproducibly score and evaluate the potential of large language models by generating test cases via table driven testing for a rudimentary and a complex example implemented in the Go programming language. The assessment focusses on the following large language models: GPT-3.5, GPT-4o, LLAMA 3, Code LLAMA and Mixtral. The evaluation highlights the ability of large language models to generate adequate test cases for low complexity test targets, while the generated test cases for high complexity test targets strongly differ in quality and tend to stagnate with faulty test cases.**

I. Introduction

Since the introduction of large language models, their potential for helping humans with repetitive tasks could potentially be large. These models, powered by advanced neural network architectures and trained on vast amounts of data, have shown remarkable capabilities in understanding and generating human-like text. Writing tests as a developer can be considered one of these repetitive tasks, often requiring significant time and effort to ensure large enough code coverage and accuracy.

In the context of software development, testing is important and helps verify the functionality and reliability of code. Despite its importance, the process of writing tests is often seen as monotonous and time-consuming. Developers need to create numerous test cases to cover various scenarios, edge cases, and potential bugs, which can be a daunting task. Here lies the potential utility of large language models: by automating the generation of test cases, these models can significantly reduce the workload on developers, allowing them to focus on more complex and creative aspects of software development.

To evaluate the concrete potential large language models have for generating tests, this paper employs different large language models and varying test targets to generate test cases in a table-driven test approach. The table-driven test approach is a systematic method where test cases are organized in a tabular format, specifying input values and the expected output for each test scenario. This approach is particularly effective for ensuring consistency and completeness in testing, making it an ideal framework for evaluating the performance of automated test generation.

The study explores multiple large language models, each with distinct architectures and training datasets, to assess their ability to generate high-quality test cases. By applying these models to different functions with differing complexities, the research aims to provide a thorough analysis of their strengths and limitations in the context of automated test generation.

Generating tests via large language models possibly could not reach the quality of hand-written tests. Human-written tests benefit from the developer's intimate understanding of the codebase, nuanced insight into potential edge cases, and the ability to apply domain-specific knowledge. Therefore, the generated output of each large language model is scored based on the code coverage.

This comparative analysis not only highlights the current capabilities of large language models but also identifies areas where they may fall short. The ultimate goal is to bridge the gap between automated test generation and human expertise, leveraging the strengths of both to enhance the software development process.

A. Theory of LLMs

LLMs (Large Language Models) are a type of artificial intelligence that can generate human-like text based on some text-input prompt. State of the art LLMs are based on the transformer architecture, which is a type of neural network that was introduced by Vaswani, Shazeer, Parmar, *et al.* in 2017.

They work by predicting a probability distribution on how likely each token is to follow the input prompt. One of the most likely token is then chosen and appended to the prompt. This process is repeated until a certain stopping criterion is met. After which the LLM has iteratively generated a text.

A general overview of the transformer architecture can be seen in figure 1. The transformer architecture consists of an encoder (left part) and a decoder (right part). The encoder processes the input prompt and the decoder generates the new output text.

The input is first converted into tokens. These are word-pieces, comparable to syllables, that are used to represent the input text. The tokens are then converted into embeddings, which are vectors in higher-dimensional space that represent the tokens. The idea behind these embeddings is that similar tokens are close to each other in this space.

The resulting values are then given some information about the current token position in the input text. This is done by

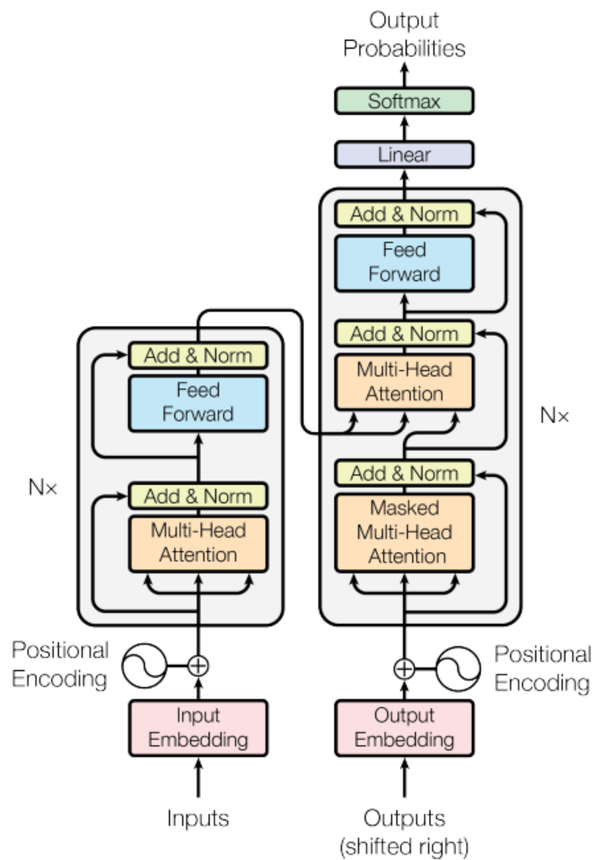

Fig. 1: The transformer architecture [1, p. 3]

adding positional encodings that represent the current position to the token embeddings. The input data is then given into the respective encoder or decoder where it is first processed using cross-attention layers. Cross-attention helps the model to focus on different parts of the input text by determining how much each token in the input text should be considered when generating the output text [2].

The output of the cross-attention layers is then processed using feed-forward neural network layers to generate a state vector. For the encoder this is passed to the next transformer block and for an encoder this state vector is used with another neural network layer to generate a probability distribution over all possible tokens using the Softmax function that scales the values to ensure that the sum of all single probabilities is 1.

From this probability distribution the next token is sampled. This can be either always the most likely token with the highest probability or by sampling from the distribution to introduce some randomness into the generation process.

The parameters for the neural network layers and the embeddings are learned using back propagation as a supervised machine learning task. For this a large text corpus is used to train the model to predict the next token in a sequence of this corpus. The performance of a LLM is impacted by the transformer architecture, size of the model, text corpus size, quality and diversity of the text corpus and the training duration. Because of this the used text corpus also has a very high impact on model performance and not only the model architecture and settings itself.

B. Problem Statement

Writing tests, test cases, and exploring edge cases may take a large amount of the time a developer invests into implementing a feature. The process of creating effective tests is not only time-consuming but also mentally taxing, requiring developers to anticipate a wide array of possible inputs and scenarios. Minimizing this impact is the target of multiple papers with differing approaches, such as [3], [4], and [5]. These studies have explored various methods to automate and streamline the testing process, leveraging the capabilities of large language models (LLMs) and other advanced techniques to reduce this burden on developers.

This paper, however, targets the gap between generating a test as a whole and generating no test at all. While previous research has demonstrated the potential of LLMs to automate test generation, there remain significant challenges with this approach. Generating a test as a whole opens the code base up to missing or incomplete tests due to the unpredictability of the output an LLM produces. This unpredictability stems from the inherent variability in the model's responses, which can lead to tests that are either overly generic, fail to cover critical edge cases, or include logical inconsistencies. As a result, developers may need to spend considerable time reviewing and refining the tests generated by LLMs, potentially offsetting the time savings these tools are intended to provide.

Addressing this issue when generating tests can save a significant amount of time and reduce the mental load on the developer implementing a feature and testing it. By focusing on more structured and reliable methods of test generation, this paper aims to bridge the gap between fully automated test creation and manual test writing. One possible approach is to integrate LLMs in a more guided and controlled manner, where the models assist developers by generating test suggestions or templates rather than complete test cases. This approach can harness the strengths of LLMs—such as their ability to quickly generate diverse ideas and scenarios—while allowing developers to maintain oversight and ensure the quality and completeness of the tests.

The aim is to develop a hybrid approach that leverages the efficiency of automation while preserving the accuracy and thoroughness of human-generated tests.

In conclusion, this research seeks to address the critical challenges in automated test generation, providing practical solutions that can be readily adopted by developers. By doing so, it aims to significantly reduce the time and effort required for testing, thereby enabling developers to focus more on innovation and feature development.

C. Objective

The goal of this paper is to evaluate the potential of large language models for test automation purposes. The research is specifically focused on comparing the performance of five

different LLMs: GPT-4o, GPT-3.5, LLAMA3, Code LLAMA and Mixtral, and a custom fine-tuned model. By leveraging the unique strengths of each model, the study aims to provide a comprehensive analysis of their capabilities in generating effective and reliable test cases.

The evaluation includes assessing the quality of their generated test cases by their coverage. These test cases are designed to test two differing functions, both implemented in the Go programming language. The selected functions represent a range of complexities, from simple arithmetic operations to more intricate data processing tasks, providing a diverse set of challenges for the LLMs to address.

The tests use a table-driven technique to ensure a structured and systematic approach to running many tests. This technique involves organizing test cases in a tabular format, specifying input values and expected outputs for each scenario. The table-driven approach is particularly effective in ensuring consistency and thoroughness, making it an ideal method for evaluating the performance of the LLMs in generating test cases.

To evaluate the generated test cases, the Go tool chain is utilized, which includes tools such as `go test` for executing tests and generating reports. This tool chain provides a robust and standardized environment for running and analyzing tests, thereby offering clear and objective insights into the capabilities and limitations of each LLM in the context of test automation. The evaluation criteria will include code coverage as the main metric.

By systematically comparing the performance of the three LLMs, this research aims to identify which model, if any, demonstrates superior capability in generating high-quality test cases. The findings of this study will contribute to a deeper understanding of the current state of LLMs in software testing and highlight areas for future improvement and research. Ultimately, the goal is to determine the feasibility of integrating LLMs into the software development life cycle to automate the testing process, thereby reducing the workload on developers and improving the efficiency and reliability of software testing.

D. Structure

At first, the theory behind large language models is introduced. This section provides an overview of the fundamental concepts, architectures, and mechanisms that large language models are built on. It covers the evolution of these models, from early natural language processing techniques to the state-of-the-art models used today, and explains key concepts such as transformer architecture.

Thereafter, the problem to solve and the objective of the paper are presented. This section defines the specific challenges and limitations associated with automated test generation in software development and outlines the goals of this research. It details the gap this study aims to address and explains why evaluating the capabilities of large language models in this context is significant.

Following, the key results of the evaluation are presented. This section showcases the findings from the empirical assessment of the three large language models used in the study. It includes quantitative metrics and qualitative observations on the performance of each model in generating test cases. Key results such as the code coverage of the tests are highlighted to provide a clear understanding of each model's strengths and weaknesses.

The methods for producing said results are then detailed. This section describes the experimental setup, including the selection of functions to be tested, the implementation of these functions in the Go programming language, and the use of a table-driven test approach. It also explains the evaluation framework, including the use of the Go tool chain to run and analyze the tests. The methodologies for scoring and comparing the generated test cases are thoroughly discussed to ensure transparency and reproducibility of the results.

A discussion of these results follows. In this section, the implications of the findings are analyzed and interpreted. The discussion addresses how the performance of the large language models aligns with or deviates from expectations, explores the potential reasons behind the observed results, and considers the broader impact of these findings on the field of automated test generation. Limitations of the study and potential avenues for future research are also discussed.

The paper ends with a summary of the paper's findings. This concluding section synthesizes the main insights gained from the research, reiterates the significance of the results, and provides final thoughts on the potential of large language models in automating test generation. It highlights the practical contributions of the study and suggests practical steps for integrating these models into the software development process to enhance efficiency and reliability.

II. Key Results

The results of the evaluation for generating test table cases using the chosen LLMs are displayed in table 2.

For the binomial coefficient function GPT-4o, LLAMA 3 and Code LLAMA are able to generate test table data that tests all cases but only the biggest model GPT-4o is able to generate them without generating any cases that fail. GPT-3.5 and Mixtral perform the worst in this comparison as they are only able to generate test tables that only test 90%/90.9% of the function but generate a failure. Since Mixtral is the smallest model and GPT-3.5 uses a rather old architecture this is to be expected.

For the MD5 algorithm test LLAMA 3, Code LLAMA and Mixtral did produce code that does not compile. The produced code has various errors like wrong imports or calling functions that do not exist showing effects of hallucinations. GPT-3.5 and GPT-4o being bigger models are not affected for this application and are able to generate valid test code. Unexpected is the fact that GPT-3.5 could generate better test cases with a coverage of 97.5% compared to GPT-4o with 96.3% that also has generated two failing test cases. Here it shows again that a bigger and more modern model does not always equate to better results due to the non-deterministic

```go
var k = []int{2, 3}
var m = map[int]string{
	k[0]: "Fizz",
	k[1]: "Buzz",
}
func FizzBuzz(n int) string {
    s := ""
    for _, key := range k {
        if n%key == 0 {
            s += m[key]
        }
    }
    if s == "" {
        return strconv.FormatInt(int64(n), 10)
    }
    return s
}
```

Listing 1: Example for a testable function with multiple branches

nature of machine learning with big neural networks, balance shifts in the used datasets, etc.

Model	Function	Score	Failed test cases
GPT-3.5	binomial	90%	2
	md5	97.5%	0
GPT-4o	binomial	100%	0
	md5	96.3%	2
LLAMA 3	binomial	100%	1
	md5	did not compile	
Code LLAMA	binomial	100%	1
	md5	did not compile	
Mixtral	binomial	90.9%	1
	md5	did not compile	

Fig. 2: Results of the LLM evaluation for Test Table Generation

III. Methods

This section lays out the methods used for coming to an evaluation of the potential LLMs can have for generating test cases in the context of test automation. The first method to be introduced is the scoring system for categorizing the output of the generated test cases. Following the criteria for selecting the assessed LLMs are presented. Subsequently factors for choosing functions to generate test cases for is explained. The closing section explains the method chosen for comparing the output of the LLMs.

A. Scoring of Generated Test Data

The go tool chain is used to score the quality of the generated test cases. Specifically, the go tool `test` accepts the `-cover` command line flag. This flag enables the computation of the coverage a test realises. Considering listing 1, the go `test` command executes the tests (see listing 3) and computes the reached coverage afterwards. The map lookup in the loop is considered a branch, as well as the number to string conversion if the string is empty. A convenient way to map these branch tests is to choose the table driven test approach, as shown in listing 3. Omitting the last case, the coverage falls from 100% to 85.7%. See listing 4 for the coverage reports. The resulting coverage value after executing all test cases a given LLM has generated is considered the score of the quality of cases the LLM has produced. It is deterministic and therefore a credible and applicable scoring value.

B. Selection of used LLMs

The proposed experiment of this paper should be evaluated with a selection of multiple LLMs. LLMs have different architectures, model sizes and used datasets that result in highly varying results depending on the used model. The used LLMs and the reasoning behind the selection are described in the following. Smaller LLMs are usually faster and more cost-effective to run, but produce lower-quality results. Since generating tests is not a time-critical task but has to be of high quality to be useful, the focus is on larger models that are known to produce better results.

The first model used in this experiment is GPT-3.5 by OpenAI which is used in the free ChatGPT version and is therefore very well known. It is based on GPT-3, a transformer-based model with 175 billion parameters and was trained on a large text corpus of the internet [6, p. 1].

The second model used is also by OpenAI and is newly released at the time of writing. It is GPT-4o [7], a multimodal model that processes text, images and sound but currently only the text modality is available. It is based on GPT-4 which is a scaled up version of GPT-3.5 but exact model details are not publicly known.

The next model used in this comparison is LLAMA 3 by Meta. It is a popular and modern open source model that can be downloaded and run by anyone in contrast to GPT-3.5 which can only be executed on the OpenAI servers. There are two variants: one with 8 billion parameters and one with 70 billion parameters [8]. For this comparison the larger model with 70 billion parameters is used. LLAMA 3 is newer and has a more advanced architecture than GPT-3.5, resulting in it being more parameter-efficient.

For LLAMA 2, the predecessor of LLAMA 3, there exists a fine-tuned version that is made especially for code, called Code LLAMA [9]. These are the based on LLAMA 2 but have another training step which includes a text corpus that only consists of code, therefore increasing performance on code specifically. For this comparison Code LLAMA Instruct with 70 billion parameters is used, the biggest model available.

The last used model is Mixtral by Mistral. It is a Mixture of Experts model that contains 8 experts in each layer. When generating text the model decides on two experts to use for each token that are most likely to work best [10]. Each expert has 7 billion parameters resulting in a total of 56 billion.

```go
package binomial

import "errors"

func BinomialCoefficient(n uint64, k uint64)
↪  (uint64, error) {
	if k > n {
		return 0, errors.New("can't
		↪  compute the binomial
		↪  coefficient for k > n")
	}

	if k == n || k == 0 {
		return 1, nil
	}
	kn := n - k
	if kn < k {
		k = kn
	}
	var r uint64 = 1
	for i := uint64(0); i < k; i++ {
		r = r * (n - i) / (i + 1)
	}
	return r, nil
}
```

Listing 2: Function for computing the binomial coefficient [11]

C. *Selection of used Exemplary Test Targets*

The functions selection process for the generation of test cases is based on the primary concern with the quality of the code and the necessity for having two functions that inhibit varying levels of complexity. The concept of complexity refers to the number of branching code paths within the function and the overall intricacy in the selected functions.

Complexity here is defined by the number of conditional statements, loops, and other control structures that determine the different execution paths the code might take. This selection aims to ensure a comprehensive evaluation of the testing framework by including functions that challenge the large language model to different extents.

For this specific purpose, two functions were chosen: one representing a lower level of complexity and the other representing a higher level of complexity. The simpler function involves the calculation of the binomial coefficient, which is a well-known mathematical function that computes the number of ways to choose k elements from a set of n elements without regard to the order. This calculation, although involving a loop, remains relatively straightforward and serves as a good baseline for comparison, see 2.

On the other hand, the more complex function is responsible for generating the `md5` hash for a given array of bytes. The `md5` algorithm processes the input data in a manner that results in a 128-bit hash value. This process involves a series of non-trivial steps, including bitwise operations and modular additions, making it considerably more complicated compared to the binomial coefficient computation.

These two functions, with their differing levels of complexity, are chosen to provide a set of test cases demonstrating the robustness and versatility of the large language model. The first function, which computes the binomial coefficient for given values of k and n, serves as the less complex example, while the second function, which generates the `md5` hash for an array of bytes, exemplifies a more complex scenario. For detailed code implementations of these functions, refer to listings and 5.

The binomial coefficient computation is taken from an open source software project implementing commonly used statistical computations [11] while the second function is part of the go standard library [12].

IV. Discussion

The results shown in chapter II are the basis for discussing the consideration of choosing a large language model for generating tests of writing the test and their cases by hand while examining the potential LLMs have for test automation.

As seen in table 2, the results differ depending on the complexity of the function to write tests for. Less complex test targets, such as the generation of test cases for the computation of the binomial coefficient being mostly successful and hitting above 90% score across all tested language models.

However the output quality rapidly declines and does not compile for larger and more complex functions, as is the case with the `md5` hashing function. This applies to smaller language models like LLAMA3, Code LLAMA and Mixtral.

The survey also assessed a code generation specific model (Code LLAMA). This model produced faulty test cases for the `md5` code example and thus code specific LLMs should not be prioritized. A better choice would be to use a generic large model like GPT-3.5 or GPT-4o. The code specific models are created for coding functions where their performance increases but this does not seem to transfer to the task of generating test cases.

A further consideration is the non-deterministic nature of LLM output, this means the developer has to check whether the test correctly tests the target while also correcting errors the LLM included in its generated output. This could potentially cancel out the time saved by not writing tests by hand.

V. Summary

This study evaluates the potential of large language models in generating test cases for software development using a table-driven testing approach for code implemented in the Go programming language. By focusing on both rudimentary and complex examples, the research assesses the capabilities of five LLMs: GPT-3.5, GPT-4o, LLAMA 3, Code LLAMA, and Mixtral. The goal is to determine the potential of these models to automate the creation of test cases, thereby reducing the workload on developers.

The evaluation reveals that while LLMs can generate adequate test cases for low-complexity targets, their performance significantly varies with higher complexity targets.

Specifically, the generated test cases for complex functions often exhibit inconsistent quality, with some models producing faulty or incomplete tests. Among the models tested, GPT-4o demonstrated the highest accuracy and coverage, particularly in generating comprehensive test cases without failures for simpler functions.

The study makes use of a scoring system based on code coverage to quantify the quality of the test cases produced by each model. The results indicate that larger, more advanced models like GPT-4o and Code LLAMA generally perform better, although challenges remain in generating reliable tests for complex functions.

In conclusion, while large language models show promise in automating test case generation, their effectiveness is currently limited by the complexity of the target functions. Further improvements in LLM architectures and training methods are necessary to improve their reliability and utility in software testing. This research contributes to the ongoing exploration of integrating AI into the software development life cycle, aiming to streamline testing processes and improve efficiency. However the non-deterministic nature of the output of large language models remains an issue stopping their inclusion in the process of testing software.

References

[1] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, *Attention is all you need*, 2017.

[2] D. Bahdanau, K. Cho, and Y. Bengio, *Neural machine translation by jointly learning to align and translate*, 2016.

[3] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: Survey, landscape, and vision," *IEEE Transactions on Software Engineering*, 2024.

[4] S. Yu, C. Fang, Y. Ling, C. Wu, and Z. Chen, "Llm for test script generation and migration: Challenges, capabilities, and opportunities," in *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*, IEEE, 2023, pp. 206–217.

[5] S. Fakhoury, A. Naik, G. Sakkas, S. Chakraborty, and S. K. Lahiri, "Llm-based test-driven interactive code generation: User study and empirical evaluation," *arXiv preprint arXiv:2404.10100*, 2024.

[6] T. B. Brown, B. Mann, N. Ryder, *et al.*, *Language models are few-shot learners*, 2020.

[7] OpenAI. "Hello GPT-4o." (2024), [Online]. Available: https://openai.com/index/hello-gpt-4o/ (visited on 05/21/2024).

[8] AI@Meta, "Llama 3 model card," 2024.

[9] B. Rozière, J. Gehring, F. Gloeckle, *et al.*, *Code llama: Open foundation models for code*, 2024.

[10] A. Q. Jiang, A. Sablayrolles, A. Roux, *et al.*, *Mixtral of experts*, 2024.

[11] xnacly. "Statlib." (Sep. 30, 2023), [Online]. Available: https://github.com/xNaCly/statlib (visited on 05/21/2024).

[12] "Md5.go." (Oct. 13, 2023), [Online]. Available: https://cs.opensource.google/go/go/+/refs/tags/go1.22.3:src/crypto/md5/md5.go;l=11 (visited on 05/21/2024).

```
$ go test -v -cover
=== RUN   TestFizzBuzz
=== RUN   TestFizzBuzz/{2_Fizz}
=== RUN   TestFizzBuzz/{12_FizzBuzz}
=== RUN   TestFizzBuzz/{1_1}
--- PASS: TestFizzBuzz (0.00s)
    --- PASS: TestFizzBuzz/{2_Fizz} (0.00s)
    --- PASS: TestFizzBuzz/{12_FizzBuzz} (0.00s)
    --- PASS: TestFizzBuzz/{1_1} (0.00s)
PASS
coverage: 100.0% of statements
ok      example 0.002s
$ go test -v -cover
=== RUN   TestFizzBuzz
=== RUN   TestFizzBuzz/{2_Fizz}
=== RUN   TestFizzBuzz/{12_FizzBuzz}
--- PASS: TestFizzBuzz (0.00s)
    --- PASS: TestFizzBuzz/{2_Fizz} (0.00s)
    --- PASS: TestFizzBuzz/{12_FizzBuzz} (0.00s)
PASS
coverage: 85.7% of statements
ok      example 0.002s
```

Listing 4: Coverage reports for the example tests

APPENDIX

```go
func TestFizzBuzz(t *testing.T) {
    cases := []struct {
        input    int
        expected string
    }{
        {2, "Fizz"},
        {12, "FizzBuzz"},
        {1, "1"},
    }

    for _, c := range cases {
        t.Run(fmt.Sprint(c), func(t *testing.T) {
            output := FizzBuzz(c.input)
            if output != c.expected {
                t.Errorf("%q != %q\n", c.expected, output)
            }
        })
    }
}
```

Listing 3: Example for the table driven testing approach for multiple branches

```go
package md5

import (
        "crypto"
        "encoding/binary"
        "errors"
        "hash"
```

```
)

func init() {
	crypto.RegisterHash(crypto.MD5, New)
}

// The size of an MD5 checksum in bytes.
const Size = 16

// The blocksize of MD5 in bytes.
const BlockSize = 64

const (
	init0 = 0x67452301
	init1 = 0xEFCDAB89
	init2 = 0x98BADCFE
	init3 = 0x10325476
)

// digest represents the partial evaluation of a checksum.
type digest struct {
	s   [4]uint32
	x   [BlockSize]byte
	nx  int
	len uint64
}

func (d *digest) Reset() {
	d.s[0] = init0
	d.s[1] = init1
	d.s[2] = init2
	d.s[3] = init3
	d.nx = 0
	d.len = 0
}

const (
	magic         = "md5\x01"
	marshaledSize = len(magic) + 4*4 + BlockSize + 8
)

func (d *digest) MarshalBinary() ([]byte, error) {
	b := make([]byte, 0, marshaledSize)
	b = append(b, magic...)
	b = binary.BigEndian.AppendUint32(b, d.s[0])
	b = binary.BigEndian.AppendUint32(b, d.s[1])
	b = binary.BigEndian.AppendUint32(b, d.s[2])
	b = binary.BigEndian.AppendUint32(b, d.s[3])
	b = append(b, d.x[:d.nx]...)
	b = b[:len(b)+len(d.x)-d.nx] // already zero
	b = binary.BigEndian.AppendUint64(b, d.len)
	return b, nil
}

func (d *digest) UnmarshalBinary(b []byte) error {
	if len(b) < len(magic) || string(b[:len(magic)]) != magic {
```

```
		return errors.New("crypto/md5: invalid hash state identifier")
	}
	if len(b) != marshaledSize {
		return errors.New("crypto/md5: invalid hash state size")
	}
	b = b[len(magic):]
	b, d.s[0] = consumeUint32(b)
	b, d.s[1] = consumeUint32(b)
	b, d.s[2] = consumeUint32(b)
	b, d.s[3] = consumeUint32(b)
	b = b[copy(d.x[:], b):]
	b, d.len = consumeUint64(b)
	d.nx = int(d.len % BlockSize)
	return nil
}

func consumeUint64(b []byte) ([]byte, uint64) {
	return b[8:], binary.BigEndian.Uint64(b[0:8])
}

func consumeUint32(b []byte) ([]byte, uint32) {
	return b[4:], binary.BigEndian.Uint32(b[0:4])
}

// New returns a new hash.Hash computing the MD5 checksum. The Hash also
// implements [encoding.BinaryMarshaler] and [encoding.BinaryUnmarshaler] to
// marshal and unmarshal the internal state of the hash.
func New() hash.Hash {
	d := new(digest)
	d.Reset()
	return d
}

func (d *digest) Size() int { return Size }

func (d *digest) BlockSize() int { return BlockSize }

func (d *digest) Write(p []byte) (nn int, err error) {
	// Note that we currently call block or blockGeneric
	// directly (guarded using haveAsm) because this allows
	// escape analysis to see that p and d don't escape.
	nn = len(p)
	d.len += uint64(nn)
	if d.nx > 0 {
		n := copy(d.x[d.nx:], p)
		d.nx += n
		if d.nx == BlockSize {
			blockGeneric(d, d.x[:])
			d.nx = 0
		}
		p = p[n:]
	}
	if len(p) >= BlockSize {
		n := len(p) &^ (BlockSize - 1)
		blockGeneric(d, p[:n])
		p = p[n:]
```

```go
	}
	if len(p) > 0 {
		d.nx = copy(d.x[:], p)
	}
	return
}

func (d *digest) Sum(in []byte) []byte {
	// Make a copy of d so that caller can keep writing and summing.
	d0 := *d
	hash := d0.checkSum()
	return append(in, hash[:]...)
}

func (d *digest) checkSum() [Size]byte {
	// Append 0x80 to the end of the message and then append zeros
	// until the length is a multiple of 56 bytes. Finally append
	// 8 bytes representing the message length in bits.
	//
	// 1 byte end marker :: 0-63 padding bytes :: 8 byte length
	tmp := [1 + 63 + 8]byte{0x80}
	pad := (55 - d.len) % 64                             // calculate number of padding bytes
	binary.LittleEndian.PutUint64(tmp[1+pad:], d.len<<3) // append length in bits
	d.Write(tmp[:1+pad+8])

	// The previous write ensures that a whole number of
	// blocks (i.e. a multiple of 64 bytes) have been hashed.
	if d.nx != 0 {
		panic("d.nx != 0")
	}

	var digest [Size]byte
	binary.LittleEndian.PutUint32(digest[0:], d.s[0])
	binary.LittleEndian.PutUint32(digest[4:], d.s[1])
	binary.LittleEndian.PutUint32(digest[8:], d.s[2])
	binary.LittleEndian.PutUint32(digest[12:], d.s[3])
	return digest
}

// Sum returns the MD5 checksum of the data.
func Sum(data []byte) [Size]byte {
	var d digest
	d.Reset()
	d.Write(data)
	return d.checkSum()
}
```

Listing 5: md5 implementation [12]

```go
// 100% 0 Fail
package binomial

func TestBinomialCoefficient(t *testing.T) {
	type input struct {
		n uint64
		k uint64
	}
```

```
	table := []struct {
		in  input
		exp uint64
		err bool
	}{
		// Test cases where k > n
		{input{5, 6}, 0, true}, // k > n

		// Test cases where k == n
		{input{5, 5}, 1, false},   // k == n
		{input{10, 10}, 1, false}, // k == n

		// Test cases where k == 0
		{input{5, 0}, 1, false},  // k == 0
		{input{10, 0}, 1, false}, // k == 0

		// Test cases for symmetry optimization (k > n/2, so k is replaced by n - k)
		{input{10, 7}, 120, false},    // 10C7 == 10C3
		{input{20, 15}, 15504, false}, // 20C15 == 20C5

		// General test cases for various n and k values
		{input{5, 2}, 10, false},       // General case
		{input{10, 2}, 45, false},      // General case
		{input{10, 3}, 120, false},     // General case
		{input{20, 10}, 184756, false}, // General case
		{input{52, 5}, 2598960, false}, // Larger values

		// Edge cases
		{input{0, 0}, 1, false}, // n == 0 and k == 0
		{input{1, 1}, 1, false}, // n == 1 and k == 1
		{input{1, 0}, 1, false}, // n == 1 and k == 0
	}
	for _, e := range table {
		t.Run(fmt.Sprint(e.in), func(t *testing.T) {
			r, err := BinomialCoefficient(e.in.n, e.in.k)
			if e.err {
				if err == nil {
					t.Errorf("expected an error for input %v, got none", e.in)
				}
			} else {
				if err != nil {
					t.Error(err)
				}
				if r != e.exp {
					t.Errorf("expected %d, got %d", e.exp, r)
				}
			}
		})

	}
}
```

Listing 6: generated binomial tests by GPT4o

```
// 90% 2 Fail
package binomial
```

```go
import (
	"fmt"
	"testing"
)

func TestBinomialCoefficient(t *testing.T) {
	type input struct {
		n uint64
		k uint64
	}
	table := []struct {
		in  input
		exp uint64
	}{
		// Cases where k > n
		{input{n: 5, k: 6}, 0},   // k > n
		{input{n: 10, k: 11}, 0}, // k > n

		// Cases where k equals n or k equals 0
		{input{n: 3, k: 3}, 1}, // k == n
		{input{n: 7, k: 0}, 1}, // k == 0

		// General cases
		{input{n: 5, k: 2}, 10}, // general case
		{input{n: 8, k: 3}, 56}, // general case
	}
	for _, e := range table {
		t.Run(fmt.Sprint(e.in), func(t *testing.T) {
			r, err := BinomialCoefficient(e.in.n, e.in.k)
			if err != nil {
				t.Error(err)
			}
			if r != e.exp {
				t.Errorf("%d != %d\n", r, e.exp)
			}
		})
	}
}
```

Listing 7: generated binomial tests by GPT3.5

```go
package binomial

import (
	"fmt"
	"testing"
)

func TestBinomialCoefficient(t *testing.T) {
	type input struct {
		n uint64
		k uint64
	}
	table := []struct {
		in  input
		exp uint64
	}{
```

```go
			{input{n: 5, k: 3}, 10},
			{input{n: 5, k: 0}, 1},
			{input{n: 5, k: 5}, 1},
			{input{n: 5, k: 6}, 0}, // error case
			{input{n: 10, k: 7}, 120},
			{input{n: 10, k: 3}, 120},
	}

	for _, e := range table {
		t.Run(fmt.Sprint(e.in), func(t *testing.T) {
			r, err := BinomialCoefficient(e.in.n, e.in.k)
			if err != nil {
				t.Error(err)
			}
			if r != e.exp {
				t.Errorf("%d != %d\n", r, e.exp)
			}
		})
	}
}
```

Listing 8: generated binomial tests by LLAMA 3