Application of Combinatorics and Python Programming in Solving Cryptarithm Problems

A Study Case of Cryptarithm War in Clash of Champions Season 1

Sophia Imelda Rogate Marpaung - 13525090
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: imeldarogate@gmail.com , 13525090@std.stei.itb.ac.id

***Abstract*—Cryptarithm is a mathematical puzzle where letters represent digits in an arithmetic equation, requiring a one-to-one letter-to-digit mapping that satisfies all arithmetic constraints. This paper discusses the application of combinatorics and Python programming to solve cryptarithm problems, using the Cryptarithm War challenge from Clash of Champions Season 1 as a study case. The mathematical modeling is based on permutations without repetition and mapping unique letters to distinct digits from 0 to 9. The backtracking algorithm systematically explores possible mappings while enforcing constraints such as no leading zeros and no repeated digit assignments. The program was tested on several competition problems. Results demonstrate that combinatorics concepts effectively model the search space while backtracking provides an efficient method for finding valid solutions automatically. This research highlights the practical application of discrete mathematics in solving real world logical and computational problems.**

Keywords—combinatorics; cryptarithm; permutations; backtracking; Python

I. Introduction

Cryptarithm is a popular mathematical puzzle in which letters are substituted for digits in an arithmetic expression. To solve the puzzle, each letter must be assigned a unique digit such that the resulting arithmetic operation is mathematically correct. Although cryptarithms appear simple, solving them manually can become increasingly difficult as the number of unique letters grows. Consequently, cryptarithms provide an interesting example of how mathematical reasoning and computational techniques can be combined to solve complex problems systematically.

One of the mathematical foundations of cryptarithm solving is combinatorics, particularly the concepts of counting rules and permutations. Since each unique letter must be assigned a distinct digit, the number of possible assignments can be modeled using permutations without repetition. However, exhaustively checking every possible assignment may require a considerable amount of computation. Therefore, an efficient search strategy is needed to reduce unnecessary evaluations.

This paper explores the application of combinatorics concepts and Python programming to solve cryptarithm problems automatically. The cryptarithm challenges featured in the Cryptarithm War of Clash of Champions Season 1 are used as case studies. The problems are modeled mathematically using permutations and injective mappings between letters and digits. Then, the backtracking algorithm is implemented in Python to search for valid solutions while satisfying all cryptarithm constraints. Through this paper, the relationship between discrete mathematics concepts and practical algorithm design is demonstrated in a real world problem solving context.

II. Theoretical Framework

A. Combinatorics

Combinatorics is a field of mathematics concerned with problems of selection, arrangement, and operation within a finite or discrete system [1]. One of the basic problems of combinatorics is to determine the number of possible configurations (*e.g* graphs, designs, arrays) of a given type. Combinatorics determines the number without having to list all possible arrangements one by one [2].

There are two basic rules that form the core concepts of permutation and combination in combinatorics.

1) The Product Rule

Suppose that when determining the total number of outcomes, two different aspects can be identified that can vary [3]. If there are n_1 possible outcomes for the first aspect, and for each of those possible outcomes, there are n_2 possible outcomes for the second aspect, then the total number of possible outcomes will be $n_1 \times n_2$.

For example, if there are 3 meals and 4 drinks to choose from, then the number of ways to choose a meal and a drink is $3 \times 4 = 12$ ways. This is true because the choices of meals and drinks are independent. The number of choices for the drink remains the same regardless of which choice was made for the meal.

2) The Sum Rule

Suppose that when determining the total number of outcomes, two distinct cases can be identified with the property that every possible outcome lies in exactly one of the cases [4]. If there are n_1 possible outcomes in the first case, and n_2 possible outcomes in the second case, then the total number of possible outcomes will be $n_1 + n_2$.

For example, if there are 3 air travel options and, distinct from them, 4 land travel options to choose from, then the number of ways to choose an air travel or a land travel option is $3 + 4 = 7$ ways. This is true because the choices are mutually exclusive. Only one method of transportation can be chosen for the trip, meaning both options cannot be taken at the same time.

B. Permutations

A permutation of n elements taken r at a time, written $P(n, r)$, is an arrangement in a row of r elements, taken from a set of n distinct elements. Unlike combinations, which ignore order, permutations place great emphasis on order.

$$P(n, r) = \frac{n!}{(n-r)!}$$

For example, in an organization, there are 10 candidates for leadership positions. Two people will be selected to serve as chair and secretary. Permutations are used to determine the number of possible leadership configurations.

$$P(10, 2) = \frac{10!}{(10-2)!} = \frac{10!}{8!} = 90$$

In addition to permutations without repetition, which are used to select r elements from a total of n available elements, permutations are also used for several other problems.

- *Permutations of n elements*: if there are n distinct elements, then the number of arrangements is

  $$P(n, n) = n!$$

  This problem is a direct application of the product rule. For example, the number of ways to arrange 3 people in 3 chairs in a row is $3! = 3 \times 2 \times 1 = 6$ ways.
- *Permutations with identical elements*: if there are several identical elements (for example, the same letters in a word), then the number of arrangements is

  $$P(n; n_1, n_2, \ldots, n_k) = \frac{n!}{n_1! \times n_2! \times \ldots \times n_k!}$$

  For example, the word MISSISSIPPI consists of 1 letter M, 4 letters I, 4 letters S, and 2 letters P. Therefore, the number of different word arrangements that can be made from the word is

  $$P(11; 1, 4, 4, 2) = \frac{11!}{1! \times 4! \times 4! \times 2!} = 34.650$$
- *Circular permutations*: if there are n distinct elements in a circular arrangement, then the number of arrangements is

  $$P_n = (n-1)!$$

  For example, the number of ways to arrange 5 people in 5 chairs surrounding a round table is

  $$P_5 = (5-1)! = 4! = 4 \times 3 \times 2 \times 1 = 24$$

C. Cryptarithm

Cryptarithm is a mathematical puzzle where the digits in a sum have been replaced by letters. The goal is to decipher the letters (*i.e* map them back onto the digits) using the constraints provided by arithmetic and the additional constraint that no two letters can have the same numerical value [5]. This type of problem was popularized during the 1930s in the *Sphinx*, a Belgian journal of recreational mathematics. To this day, this problem is used in schools to improve students' ability to add integers and encourage them to select possible integers based on their properties [6]. Students will practice mathematical reasoning and systematic working.

In cryptarithm, addition operations use a decimal system with a maximum of ten unique letters, where each letter represents a different digit (0-9) and the first letter of a word cannot be zero. For a solution to be considered correct, substituting these letters must result in a valid mathematical calculation consistent with the specified operation.

Here is one of the famous cryptarithm problems which was published in the July 1924 issue of Strand Magazine by Henry Dudeney.

```
  SEND
  MORE
 ----- +
 MONEY
```

Fig. 1. Cryptarithm problem example.

Assigning these specific digits to the letters provides a mathematically correct solution to the puzzle.

O = 0, M = 1, Y = 2, E = 5, N = 6, D = 7, R = 8, S = 9

```
  9567
  1085
 ----- +
 10652
```

Fig. 2. An acceptable solution for the problem in Fig. 1.

III. METHODOLOGY

A. Research Design

This research employs a descriptive qualitative method with a case study approach. It focuses on analyzing and modeling specific cryptarithm problems from a real-world event, combined with an applied computational approach (algorithm design) to solve the problems using Python programming. Python was chosen as the implementation language due to its clean and readable syntax, which makes the logic of the backtracking algorithm easy to follow and understand. In addition, Python provides built-in modules such

as *math* and *time* that directly support the needs of this program, eliminating the need for external dependencies.

B. Data Collection

Data collection was conducted by watching the replay of Clash of Champions Season 1 Episode 4, which was aired on YouTube on July 7, 2024. In episodes 4 and 5, cryptarithm problems were one of the game themes featured in the competition, specifically the Cryptarithm War.

Players were challenged to solve the provided cryptarithm problems quickly and accurately. A board consisting of 20 boxes arranged in 4 rows and 5 columns was provided. Each box contained a cryptarithm problem that had to be solved. These are 10 cryptarithm problems that were successfully collected as data for this research.

Problem A1	Problem B1	Problem C1	Problem D1	Problem E1
I I	A R A	A B	A B	A I R
I I	A B A	A B	B A	A R
----- +	----- +	----- +	----- +	----- +
H I U	B A R	B C C	D A D	I R A

Problem A2	Problem B2	Problem C2	Problem D2	Problem E2
A V I	A B B	A M	A A A	I M A
V I	A A	P M	A B	A I M
----- +	----- -	----- +	----- -	------- +
I A N	B A C	A L L	C C	M A K I

Fig. 3. Cryptarithm War problems from Clash of Champions Season 1 Episode 4.

Those cryptarithm problems involve not only addition but also subtraction, as seen in problems B2 and D2 in Fig. 3.

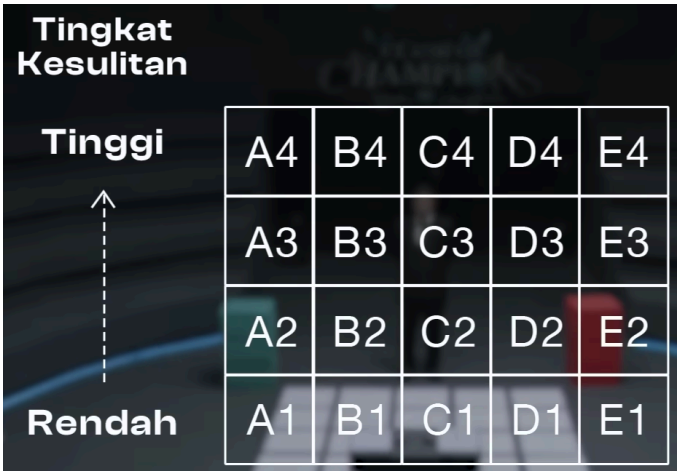

Fig. 4. The difficulty level of the cryptarithm problems to be solved.
(Source: https://youtu.be/_2yf29Wdekc?si=Nc-ay1HLNN3xoAPs)

As shown in Fig. 4, problems A1 through E1 are less difficult than problems A2 through E2, and so on. This difficulty level is determined based on the number of variables (unique letters), patterns of symmetry and letter repetition, and the variety of arithmetic operations.

C. Mathematical Modeling

In this research, cryptarithm is modeled as combinatorics problems that also involve mapping a set of letters to a set of digits. This mapping can be expressed as a one-to-one or injective function.

Let L be the set of distinct letters that appear in a cryptarithm problem.

$$L = \{l_1, l_2, ..., l_k\}$$

Each letter is mapped to exactly one distinct element of the set D, which is the set of integers from 0 to 9. Therefore, the cardinality of set D is 10.

$$D = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

However, it should be noted that the letter at the very beginning of a word in a cryptarithm problem must not be mapped to the digit 0, so as not to produce a number that begins with zero.

The number of possible mappings to be tested is determined using the concept of permutations without repetition. If there are r unique letters that must be mapped to one of the 10 digits, then the number of possibilities is

$$P(10, r) = \frac{10!}{(10-r)!}$$

Each possibility is then tested against the arithmetic operations in the cryptarithm equation. If the result of the addition or subtraction operation matches, that possibility is immediately declared the correct solution (i.e., there is no need to consider other possibilities). The mathematical model for addition is expressed as $n_1 + n_2 = n_3$ while the model for subtraction is expressed as $n_1 - n_2 = n_3$ where n_1, n_2, and n_3 are the numerical values obtained after all letters are replaced with the corresponding digits.

D. Algorithm Design

The algorithm used in this program is the backtracking algorithm, which systematically explores possible digit assignment for each unique letter without evaluating the entire equation until all letters have been mapped. Starting with the first letter in the list, the algorithm recursively tries every digit from 0 to 9, applying the constraints defined in the mathematical model. No digit may be reused for a different letter and no letter at the beginning of a word may be mapped to the digit 0. If all letters have been mapped to digits, the equation is evaluated according to the selected operator. If the mapping satisfies the equation, the solution is recorded and the search ends immediately. If not, the algorithm performs backtracking by discarding the current mapping and trying the next available digit. This recursive process continues until a valid solution is found or all possible P(10, r) combinations have been exhausted without finding a valid mapping.

IV. Implementation

A. Algorithm Support Tools

This script starts by importing the *factorial* function from the *math* module, which is part of Python's standard library.

In addition, the *time* module is also imported to measure the program's execution time so that users can compare the time it takes a person to solve a cryptarithm problem with the time it takes this program to do so.

```python
from math import factorial
import time

def permutation_count(n, r):
    return factorial(n) // factorial(n - r)

def word_to_number(word, mapping):
    number = ""
    for letter in word:
        number += str(mapping[letter])
    return int(number)
```

Fig. 5. Algorithm Support Tools

The $permutation\_count$ function calculates the number of permutations without repetition, $P(n, r)$, which is used to determine the total number of possible digit combinations that need to be checked. The $word\_to\_number$ function converts a word into a number based on the given letter-to-digit mapping. This function iterates through each letter in the word, retrieves the corresponding digit from the $mapping$, and then combines them into a string of numbers, which is finally converted to an int and returned as the result.

B. Initialization and Unique Letter Set Construction

```python
class CryptarithmSolver:
    def __init__(self, word1, word2, result, operator):
        self.word1 = word1.upper()
        self.word2 = word2.upper()
        self.result = result.upper()
        self.operator = operator

        self.letters = []
        for ch in self.word1 + self.word2 + self.result:
            if ch not in self.letters:
                self.letters.append(ch)

        self.first_letters = {
            self.word1[0],
            self.word2[0],
            self.result[0]
        }

        self.mapping = {}
        self.used_digits = set()
        self.solution_found = False
        self.candidates_checked = 0
```

Fig. 6. Initialization and Unique Letter Set Construction

The $CryptarithmSolver$ class is created with a $\_\_init\_\_$ constructor that accepts four inputs, that is the two words to be operated on ($word1$, $word2$), the result word ($result$), and the operator ($+$ or $-$). All three words are immediately converted to uppercase using $.upper()$ so that the letter checks are not case-sensitive. After that, the program collects all unique letters from the three words into the $self.letters$ list by iterating through the letters one by one and adding them only if they are not already in the list. Next, $self.first\_letters$ stores the first letter of each word as a set, which will later be used to ensure that the first letter cannot be mapped to the digit 0. Finally, several other supporting attributes also need to be initialized. $self.mapping$ is initialized as an empty dictionary to store letter to digit pairs, $self.used\_digits$ as an empty set to track digits that have already been used, $self.solution\_found$ is set to $False$ to indicate whether a solution has been found, and $self.candidates\_checked$ is set to 0 to count how many combinations have been tested.

C. Cryptarithm Solution Verification

```python
    def check_solution(self):
        self.candidates_checked += 1

        n1 = word_to_number(self.word1, self.mapping)
        n2 = word_to_number(self.word2, self.mapping)
        n3 = word_to_number(self.result, self.mapping)

        valid = False

        if self.operator == "+":
            valid = (n1 + n2 == n3)
        elif self.operator == "-":
            valid = (n1 - n2 == n3)

        if valid:
            self.solution_found = True
            print("Solution Found!")

            for letter in sorted(self.mapping):
                print(f"{letter} = {self.mapping[letter]}")

            print("")

            width = max(len(str(n1)), len(str(n2)), len(str(n3)))
            print(str(n1).rjust(width))
            print(str(n2).rjust(width))
            print("-" * width + f" {self.operator}")
            print(str(n3).rjust(width))

            print("\nNumber of possibilities checked :",
                  self.candidates_checked)
```

Fig. 7. Cryptarithm Solution Verification

The $check\_solution$ function is responsible for checking whether the combination of digits currently being tested is a valid solution. Each time this function is called, $self.candidates\_checked$ is incremented by one to record that one candidate has been checked. The three words are converted to numbers using the $word\_to\_number$ function based on the currently active $self.mapping$, and the results are stored in n_1, n_2, and n_3. The $valid$ variable is initialized to $False$ and then checked according to the operator. If the operation is addition ($+$), the program checks whether $n_1 + n_2 == n_3$. If it is subtraction ($-$), it checks whether $n_1 - n_2 == n_3$. If $valid$ is $True$, a solution has been found. $self.solution\_found$ is set to $True$ and the program prints the letter-to-digit mapping in alphabetical order using $sorted(self.mapping)$. After that, the program displays a visual representation of the equation using $width$ and $.rjust(width)$ to ensure a neat layout.

D. Backtracking Algorithm Implementation

```
def backtrack(self, index):
    if self.solution_found:
        return

    if index == len(self.letters):
        self.check_solution()
        return

    current_letter = self.letters[index]

    for digit in range(10):
        if digit in self.used_digits:
            continue

        if current_letter in self.first_letters and digit == 0:
            continue

        self.mapping[current_letter] = digit
        self.used_digits.add(digit)
        self.backtrack(index + 1)
        self.used_digits.remove(digit)
        del self.mapping[current_letter]
```

Fig. 8. Backtracking Algorithm Implementation

The *backtrack* function is responsible for recursively testing all possible combinations of digits. If a solution has been found, the function stops immediately to avoid wasting time trying other combinations. If the index is equal to the number of unique letters, then each letter has been assigned a digit, so *check_solution* is called to verify whether this combination is valid. If not, the program retrieves the character currently being processed via $self.letters[index]$ and attempts to assign digits 0 through 9 one by one, skipping any that violate the constraints defined in the Algorithm Design section. If the digit passes both checks, it is assigned to the letter via $self.mapping$, recorded in $self.used\_digits$, and *backtrack* is called again with $index + 1$ to process the next letter. Once the recursion completes or returns, the digit is removed from $self.mapping$ and $self.used\_digits$.

E. Solution Process Using Permutation and Backtracking

```
def solve(self):
    print("\nPROBLEM INFORMATION")
    print("Unique letters :", self.letters)
    r = len(self.letters)
    print("Number of unique letters :", r)

    if r > 10:
        print("\nError: The number of unique letters exceeds
            10!")
        return

    total = permutation_count(10, r)
    print(f"Total number of possibilities = P(10, {r}) =
        {total}")

    print("\nSearching for a solution...\n")
    start = time.time()
    self.backtrack(0)
    end = time.time()

    if not self.solution_found:
        print("No solution found!")

    print("Execution time : %.3f seconds" % (end - start))
```

Fig. 9. Solution Process Using Permutation and Backtracking

The *solve* function is the main function called to execute the entire solution-search process. The program prints initial information in the form of a list of unique letters found and their count. Before continuing, the program checks whether the number of unique letters exceeds 10. If so, the program prints an error message and immediately terminates using *return*. If the number of unique letters is valid, the program calculates the total number of possible combinations using $permutation\_count(10, r)$. After the information is displayed, the program records the start time using $time.time()$ and then calls $self.backtrack(0)$. Once the backtracking is complete, the end time is recorded. If after the entire process $self.solution\_found$ is still $False$, the program prints a message stating that no solution was found. In the final step, the difference between the end time and the start time is printed as the execution duration so that the user can see how long the program took to find a solution.

F. Main Program Execution and Result Analysis

Main program section is the part that directly interacts with the user. The program begins by printing the title, then prompts the user to enter three words one after another (first word, second word, and result).

```
# Main Program
print("CRYPTARITHM SOLVER PROGRAM\n")

while True:
    word1 = input("Enter the first word : ").strip()
    if word1.isalpha():
        break
    print("Error: Words must contain only letters!\n")

while True:
    word2 = input("Enter the second word : ").strip()
    if word2.isalpha():
        break
    print("Error: Words must contain only letters!\n")

while True:
    result = input("Enter the result : ").strip()
    if result.isalpha():
        break
    print("Error: Words must contain only letters!\n")

while True:
    operator = input("Select the operator + or - : ").strip()
    if operator in ["+", "-"]:
        break
    print("Error: Valid operators are only + or - !\n")

solver = CryptarithmSolver(
    word1,
    word2,
    result,
    operator
)

solver.solve()
```

Fig. 10. Main Program Execution and Result Analysis

Each input uses a *while True* loop that continues to prompt for input until the user enters a valid value. Validation is performed using $isalpha()$, which ensures that the input consists only of letters. Inputs containing numbers, spaces, and symbols are rejected, and the program prints an error message before requesting input again. The same applies to operators;

inputs are only accepted if they are + or −. Additionally, $.strip()$ is used to remove any spaces that may have been accidentally typed at the beginning or end of the input. Once all inputs have been collected, a $CryptarithmSolver$ object is created with the four inputs as arguments, which automatically calls $\_\_init\_\_$ to initialize all required attributes. At the end, $solver.solve()$ is called to start the entire process of finding the solution.

The next step is to run the cryptarithm solver program in the terminal so that all the cryptarithm problems in Fig. 3 can be solved.

```
CRYPTARITHM SOLVER PROGRAM

Enter the first word : ARA
Enter the second word : ABA
Enter the result : BAR
Select the operator + or - : +

PROBLEM INFORMATION
Unique letters : ['A', 'R', 'B']
Number of unique letters : 3
Total number of possibilities = P(10, 3) = 720

Searching for a solution...

Solution Found!
A = 3
B = 7
R = 6

363
373
--- +
736

Number of possibilities checked : 169
Execution time : 0.002 seconds
```

Fig. 11. Program Execution Results for Problem B1

```
CRYPTARITHM SOLVER PROGRAM

Enter the first word : ABB
Enter the second word : AA
Enter the result : BAC
Select the operator + or - : -

PROBLEM INFORMATION
Unique letters : ['A', 'B', 'C']
Number of unique letters : 3
Total number of possibilities = P(10, 3) = 720

Searching for a solution...

Solution Found!
A = 8
B = 7
C = 9

877
 88
--- -
789

Number of possibilities checked : 504
Execution time : 0.003 seconds
```

Fig. 12. Program Execution Results for Problem B2

After implementing all the cryptarithm problems in Fig. 3 into the cryptarithm solver program, the solutions are shown in Fig. 13.

Solution A1	Solution B1	Solution C1	Solution D1	Solution E1
9 9	3 6 3	6 1	2 9	6 7 3
9 9	3 7 3	6 1	9 2	6 3
----- +	----- +	----- +	----- +	----- +
1 9 8	7 3 6	1 2 2	1 2 1	7 3 6

Solution A2	Solution B2	Solution C2	Solution D2	Solution E2
5 7 6	8 7 7	1 5	1 1 1	9 1 8
7 6	8 8	8 5	1 2	8 9 1
----- +	----- -	----- +	----- -	------- +
6 5 2	7 8 9	1 0 0	9 9	1 8 0 9

Fig. 13. All acceptable solutions for problems in Fig. 3.

V. Conclusion

This paper demonstrates that cryptarithm problems can be effectively modeled and solved using concepts from combinatorics and Python programming. The number of possible letter to digit assignments can be represented using permutations without repetition, while the constraints of cryptarithm problems can be expressed through an injective mapping between letters and digits. Based on this mathematical model, the backtracking algorithm was implemented in Python to systematically search for valid solutions.

The experimental results show that the developed program is capable of solving the cryptarithm problems collected from the Cryptarithm War challenge in Clash of Champions Season 1. By combining combinatorics counting with recursive search techniques, the program can efficiently identify valid assignments without requiring manual trial and error. Therefore, this paper shows how discrete mathematics concepts, particularly combinatorics and permutations, can be applied to practical computational problems and implemented through algorithmic programming.

Attachment

Cryptarithm solver program source code Here
Paper explanation video Here

Acknowledgment

The author offers praise and thanks to God Almighty. Through His blessings and graces, the author was able to complete this paper. The author would like to thank Prof. Dr. Ir. Rinaldi, M. T., for his guidance and support throughout this semester as the lecturer for the Discrete Mathematics course. The author would also like to thank family and friends who consistently provided motivation and encouragement throughout the course of study. Finally, the author hopes this

paper will contribute to the field of education for the community.

References

[1] R. C. Bose, "Combinatorics," Encyclopaedia Britannica, Apr. 25, 2026. [Online]. Available: https://www.britannica.com/science/combinatorics. [Accessed: Jun. 14, 2026].

[2] R. Munir, "Kombinatorika bagian 1," School of Electrical Engineering and Informatics, Institut Teknologi Bandung, Bandung, Indonesia, Lecture Notes, 2026. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/18-Kombinatorika-Bagian1-2026.pdf. [Accessed: Jun. 14, 2026].

[3] J. Morris, "The product rule," in Combinatorics. LibreTexts, 2021. [Online]. Available: https://math.libretexts.org/Bookshelves/Combinatorics_and_Discrete_Mathematics/Combinatorics_(Morris)/02%3A_Enumeration/02%3A_Basic_Counting_Techniques/2.01%3A_The_Product_Rule. [Accessed: Jun. 15, 2026].

[4] J. Morris, "The sum rule," in Combinatorics. LibreTexts, 2021. [Online]. Available: https://math.libretexts.org/Bookshelves/Combinatorics_and_Discrete_Mathematics/Combinatorics_(Morris)/02%3A_Enumeration/02%3A_Basic_Counting_Techniques/2.02%3A_The_Sum_Rule. [Accessed: Jun. 15, 2026].

[5] NRICH, "Cryptarithms," University of Cambridge. [Online]. Available: https://nrich.maths.org/problems/cryptarithms. [Accessed: Jun. 16, 2026].

[6] R. Abbasian and M. Mazloom, "Solving cryptarithmetic problems using parallel genetic algorithm," in Proc. 2nd Int. Conf. Comput. Electr. Eng. (ICCEE), vol. 1, pp. 308-312, Dec. 2009.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 19 Juni 2026

Sophia Imelda Rogate Marpaung
13525090