VIETNAM GENERAL CONFEDERATION OF LABOR
TON DUC THANG UNIVERSITY
FACULTY OF INFORMATION TECHNOLOGY

NGUYEN QUANG TRUONG – 523H0110
CHUNG QUANG VU – 523H0196

MIDTERM REPORT

DISCRETE STRUCTURES

HO CHI MINH CITY, 2025

VIETNAM GENERAL CONFEDERATION OF LABOR

TON DUC THANG UNIVERSITY

FACULTY OF INFORMATION TECHNOLOGY

NGUYEN QUANG TRUONG – 523H0110

CHUNG QUANG VU – 523H0196

MIDTERM REPORT

DISCRETE STRUCTURES

Advised by

Dr. Nguyen Quoc Binh

HO CHI MINH CITY, 2025

ACKNOWLEDGEMENT

We sincerely appreciate the invaluable support and assistance you have offered during the preparation of this midterm essay. Our heartfelt thanks go to Dr. Nguyen Quoc Binh for his unwavering guidance and the insightful knowledge he shared, which greatly contributed to the successful completion of this essay. His mentorship and constructive feedback have played a crucial role in deepening our understanding and improving our skills in Discrete Structures.

Ho Chi Minh city, 2nd May 2025.

Author

(Signature and full name)

Truong

Nguyen Quang Truong

Vu

Chung Quang Vu

DECLARATION OF AUTHORSHIP

We hereby declare that this is our own project and is guided by Dr. Nguyen Quoc Binh; The content research and results contained herein are central and have not been published in any form before. The data in the tables for analysis, comments and evaluation are collected by the main author from different sources, which are clearly stated in the reference section.

In addition, the project also uses some comments, assessments as well as data of other authors, other organizations with citations and annotated sources.

If something wrong happens, we'll take full responsibility for the content of our project. Ton Duc Thang University is not related to the infringing rights, the copyrights that We give during the implementation process (if any).

Ho Chi Minh city, 2nd May 2025.

Author

(Signature and full name)

Truong

Nguyen Quang Truong

Vu

Chung Quang Vu

TABLE OF CONTENT

LIST OF IMAGES

LIST OF TABLE

LIST OF ABBREVIATIONS

RPN	Reverse Polish Notation
RSA	Rivest–Shamir–Adleman
CRT	Chinese Remainder Theorem
OAEP	Optimal Asymmetric Encryption Padding
SHA	Secure Hash Algorithm
AES	Advanced Encryption Standard
PKCS	Public Key Cryptography Standard
TLS	Transport Layer Security
SSL	Secure Sockets Layer
DHE	Diffie-Hellman Ephermal
HSM	Hardware Security Module
TPM	Trusted Platform Module
PQC	Post-Quantum Cryptography
VPN	Virtual Private Network

CHAPTER 1. TASK ASSIGNMENT AND SELF-EVALUATION.

No.	Full Name	Student ID	Assigned task	Self-evaluation
1	Nguyen Quang Truong	523H0110	Task 1	100%
			Report	100%
2	Chung Quang Vu	523H0196	Task 3	100%
			Task 2	100%

Table 1 Task assignment and self-evaluation

CHAPTER 2. TRUTH TABLE

2.1 Theoretical Basis

2.1.1 Reverse Polish

2.1.1.1 Definition

Reverse Polish Notation (RPN)—also called reverse Lukasiewicz notation, Polish postfix notation, or simply postfix notation—is a mathematical notation in which each operator follows its operands. In contrast, prefix (Polish) notation places operators before their operands. Because every operator in RPN has a fixed arity, no parentheses are ever required: the order of operations is determined entirely by the sequence of tokens.

2.1.1.2 Advantages

Parentheses-free: Operator precedence is handled implicitly by token order.
Stack-friendly: Ideal for stack-based evaluation: operands are pushed onto a stack; when an operator appears, the required operands are popped off, the operation is performed, and the result is pushed back.

2.1.1.3 Operational Mechanism (stack-based evaluation)

- Read tokens left to right.
- Operand → push onto stack.
- Operator → pop the necessary operands, apply the operation, push the result.
- End of expression → exactly one value remains on the stack (the final result).

Example:

- Infix: (3 + 5)   7
- Postfix (RPN): 3 5 + 7

2.1.2 Basic logic used on calculation of Truth tables

2.1.2.1 Operator precedence for logical operators

Operator Name	Symbol(s)	Meaning
NOT(Negation)	~A, ¬A	Reverses the logic value: true → false, false → true.
AND(Conjunction)	A ∧ B, A && B	True only if both A and B are true.
OR(Disjunction)	A ∨ B, A || B	Returns true if at least one of the values is true. It only returns false when both values are false.
Implication	A → B	False only when A is true and B is false; true in all other cases.
Biconditional	A ↔ B	True when A and B have the same logic value (both true or both false).

Table 2 Operator precedence for logical operators

2.1.2.2 Truth table logic

P	**Q**	**¬P**	**P ∧ Q**	**P ∨ Q**	**P → Q**	**P ↔ Q**
T	T	F	T	T	T	T
T	F	F	F	T	F	F
F	T	T	F	T	T	F
F	F	T	F	F	T	T

Table 3 Truth table logic

2.2 Program explanation

2.2.1 Function Infix2Postfix(Infix)

General description: We first define operator precedence and note that “not” is right-associative. We also initialize an empty list for output tokens and an empty stack for operators. Next, we traverse the input expression. For each token if it's an operand (A–Z), we append it directly to the output list. If it's a left parenthesis ‘(‘, we push it onto the stack. If it's a right parenthesis ‘)', we pop operators from the stack onto the output list until we reach the matching ‘(‘, which we then discard. Otherwise, it's an operator: we pop any operators of higher precedence—or of equal precedence if the current operator is left-associative—from the stack onto the output list, then push the current operator onto the stack. Finally, after processing all tokens, we pop any remaining operators from the stack onto the output list and return the concatenated postfix string.

2.2.1.1 Test case 1 (R|(P&Q))

Initialize two arrays output and stack then we traversal the input R|(P&Q)

Step 1: token = 'R' is operand ⇒ append to output so output = ['R'], stack = []

Step 2: token = '|' is operator ⇒ push to stack so output = ['R'] , stack = ['|']

Step 3: token = '(' is opening parenthesis ⇒ push to stack so output = ['R'], stack = ['|', '(']

Step 4: token = 'P' is operand ⇒ append to output so output = ['R', 'P'], stack = ['|', '(']

Step 5: token = '&' is operator ⇒ push to stack (because top is '(', no precedence comparison) so output = ['R', 'P'], stack = ['|', '(', '&']

Step 6: token = 'Q' is operand ⇒ append to output so output = ['R', 'P', 'Q'], stack = ['|', '(', '&']

Step 7: token = ')', Pop for until '(' is encountered, Pop '&' → append → output = ['R', 'P', 'Q', '&'], Meet '(' → pop but don't append, stack = ['|']

Finally: Pop the entire stack output set. Pop '|' → output = ['R', 'P', 'Q', '&', '|'], stack = []

Result: RPQ&|

2.2.1.2 Test case 2 (~P|(Q&R)>R)

Initialize two arrays output and stack, then traverse the input ~P|(Q&R)>R.

Step 1: token = ‘~’ is an operator, top is ‘~’ of equal precedence and right-associative ⇒ don't pop, push to stack → stack = ['~'].

Step 2: token = ‘P’ is an operand ⇒ append to output → output = ['P'], stack = ['~'].

Step 3: token = ‘|’ is an operator; top = ‘~’ has higher precedence ⇒ pop ‘~’ → output = ['P','~']; pop next ‘~’ → output = ['P','~']; then push ‘|’ → stack = ['|'].

Step 4: token = ‘(‘ is a left parenthesis ⇒ push → stack = ['|','('].

Step 5: token = ‘Q’ is an operand ⇒ append → output = ['P','~','Q'], stack = ['|','('].

Step 6: token = ‘&’ is an operator; top is ‘(‘ ⇒ push → stack = ['|','(','&'].

Step 7: token = ‘R’ is an operand ⇒ append → output = ['P','~','Q','R'], stack = ['|','(','&'].

Step 8: token = ‘)’ ⇒ pop stack until meet ‘(‘ ⇒ pop ‘&’ → output = ['P','~','Q','R','&']; discard ‘(’ → stack = ['|'].

Step 9: token = ‘>’ is an operator; top = ‘|’ has higher precedence ⇒ pop ‘|’ → output = ['P','~','Q','R','&','|'], then push ‘>’ → stack = ['>'].

Step 10: token = ‘R’ is an operand ⇒ append → output = ['P','~','Q','R','&','|','R'].

Finally, pop all remaining operators: pop > → output = ['P','~','Q','R','&','|','R','>'], yielding the postfix string P~QR&|R>.

2.2.2 Function Postfix2Truthtable(Postfix)

General description: We define a function that takes a postfix logical expression and prints its full truth table. First, it extracts all unique variable names (A–Z) from the expression, sorts them, and prints a header row listing each variable followed by the expression itself. It then iterates over every possible combination of truth values for those variables (using Cartesian product of True/False). For each combination, it builds an environment mapping variables to their current truth values and evaluates the postfix expression using a stack: pushing variable values, applying unary NOT by popping one value and pushing its negation, and applying binary operators (AND, OR, IMPLIES, IFF) by popping two values (right operand first), computing the result, and pushing it back. After processing all tokens, the single remaining stack value is the expression’s truth value for that assignment, which it prints alongside the variable values.

2.2.2.1 Test case 1 (R|(P&Q))

Postfix: RPQ&|

Variables (sorted): ['P', 'Q', 'R']

Number of vars: 3 → $2^3 = 8$ combinations

All combinations and their env dictionaries:

P	Q	R	Dictionary
True	True	True	{'P': True, 'Q': True, 'R': True}
True	True	False	{'P': True, 'Q': True, 'R': False}
True	False	True	{'P': True, 'Q': False, 'R': True}
True	False	False	{'P': True, 'Q': False, 'R': False}
False	True	True	{'P': False, 'Q': True, 'R': True}
False	True	False	{'P': False, 'Q': True, 'R': False}
False	False	True	{'P': False, 'Q': False, 'R': True}
False	False	False	{'P': False, 'Q': False, 'R': False}

Table 4 All possible combinations and their dictionary

This corresponds to the for values in itertools.product([True, False], repeat=num_vars): loop and env = dict(zip(variables, values)).)

Example evaluation for P=True, Q=False, R=True. Postfix tokens: ['R','P','Q','&','|']

Token	Stack before	Action	Stack after
R	[]	token.isalpha() → push env['R'] = True	[True]
P	[True]	push env['P'] = True	[True, True]
Q	[True, True]	push env['Q'] = False	[True, True, False]
&	[True, True, False]	binary AND → pop right=False, left=True → True and False = False, push False	[True, False]
|	[True, False]	binary OR → pop right=False,left=True → True or False = True, push True	[True]

Table 5 Explanation of Postfix2Truthtable(Postfix) test case 1

→ Result = stack.pop() = True

Final truth table (printed by each print(' | '.join(row)))

P	Q	R	Result
True	True	True	True
True	True	False	True
True	False	True	True
True	False	False	False
False	True	True	True
False	True	False	False
False	False	True	True
False	False	False	False

Table 6 Result of all combinations of test case 1

2.2.2.2 Test case 2 (~P|(Q&R)>R)

Postfix: P~QR&|R>|

Variables (sorted): ['P', 'Q', 'R']

Number of vars: 3 → $2^3 = 8$ combinations

All combinations and their env dictionaries:

P	Q	R	Dictionary
True	True	True	{'P': True, 'Q': True, 'R': True}
True	True	False	{'P': True, 'Q': True, 'R': False}
True	False	True	{'P': True, 'Q': False, 'R': True}
True	False	False	{'P': True, 'Q': False, 'R': False}
False	True	True	{'P': False, 'Q': True, 'R': True}
False	True	False	{'P': False, 'Q': True, 'R': False}
False	False	True	{'P': False, 'Q': False, 'R': True}
False	False	False	{'P': False, 'Q': False, 'R': False}

Table 7 All possible combinations and their dictionary

Example evaluation for P=True, Q=False, R=True

Token	Stack before	Action	Stack after
P	[]	token.isalpha() → push env['P'] = True	[True]
~	[True]	unary NOT → pop True, push not True = False	[False]
Q	[False]	push env['Q'] = False	[False, False]
R	[False, False]	push env['R'] = True	[False, False, True]
&	[False, False, True]	pop right=True, left=False → False and True = False; push False	[False, False]
|	[False, False]	pop right=False, left=False → False or False = False; push False	[False]
R	[False]	push env['R'] = True	[False, True]

>	[False, True]	implication → pop right=True, left=False → (not False) or True = True or True = True; push True	[True]

Table 8 Explanation of Postfix2Truthtable(Postfix) test case 2

→ Result = stack.pop() = True

Final truth table (printed by each print(' | '.join(row)))

P	**Q**	**R**	**Result**
True	True	True	True
True	True	False	True
True	False	True	True
True	False	False	True
False	True	True	True
False	True	False	False
False	False	True	True
False	False	False	False

Table 9 Result of all combinations of test case 2

2.3 Result of 5 test cases

```
Infix: R|(P&Q)
Postfix: RPQ&|
P | Q | R | RPQ&|
-------------
True | True | True | True
True | True | False | True
True | False | True | True
True | False | False | False
False | True | True | True
False | True | False | False
False | False | True | True
False | False | False | False
```

Image 1. Test case 1

```
Infix: ~P|(Q&R)>R
Postfix: P~QR&|R>
P | Q | R | P~QR&|R>
-------------
True | True | True | True
True | True | False | True
True | False | True | True
True | False | False | True
False | True | True | True
False | True | False | False
False | False | True | True
False | False | False | False
```

Image 2 Test case 2

```
Infix: P|(R&Q)
Postfix: PRQ&|
P | Q | R | PRQ&|
-------------
True | True | True | True
True | True | False | True
True | False | True | True
True | False | False | True
False | True | True | True
False | True | False | False
False | False | True | False
False | False | False | False
```

Image 3 Test case 3

```
Infix: (P>Q)&(Q>R)
Postfix: PQ>QR>&
P | Q | R | PQ>QR>&
-------------
True | True | True | True
True | True | False | False
True | False | True | False
True | False | False | False
False | True | True | True
False | True | False | False
False | False | True | True
False | False | False | True
```

Image 4 Test case 4

```
Infix: (P|~Q)>~P=(P|(~Q))>~P
Postfix: PQ~|P~>PQ~|P~>=
P | Q | PQ~|P~>PQ~|P~>=
---------
True | True | True
True | False | True
False | True | True
False | False | True
```

Image 5 Test case 5

CHAPTER 3. QUANTIFIED REASONING OVER REAL-WORLD DATA USING PREDICATE LOGIC

3.1 Create dataset

To carry out quantified reasoning, I created a dataset called students.csv with 20 entries. Each entry includes the following fields:

StudentID: a unique ID for each student,

StudentName: the student's name,

DayOfBirth: date of birth,

Math, CS, Eng: scores in three subjects (Math, Computer Science, English).

The scores were randomly generated to mimic realistic academic performance. This dataset serves as the foundation for evaluating predicates.

Example:

Data from file CSV:

	StudentID	StudentName	DayOfBirth	Math	CS	Eng
0	SV001	Do Mai	31/05/2001	9.3	8.1	10.0
1	SV002	Tran Nam	23/06/2002	9.5	9.4	8.5
2	SV003	Nguyen Mai	04/02/2002	8.2	8.0	8.3
3	SV004	Do Hai	25/01/2003	9.8	8.3	9.8
4	SV005	Dang Anh	18/10/2004	9.6	9.2	9.4
5	SV006	Tran Mai	07/06/2003	9.6	7.1	6.7
6	SV007	Nguyen Duc	02/04/2005	9.7	7.0	7.3
7	SV008	Pham Tuan	04/01/2001	9.4	7.0	7.3
8	SV009	Pham Anh	17/06/2004	9.8	7.3	6.7
9	SV010	Bui Tuan	10/12/2000	9.3	7.9	6.7
10	SV011	Vo Lan	11/09/2004	6.4	7.8	5.3
11	SV012	Tran Tuan	12/12/2002	4.9	7.7	8.8
12	SV013	Tran Minh	19/10/2005	5.6	7.3	5.4
13	SV014	Pham Nam	03/06/2000	4.6	6.7	5.5
14	SV015	Dinh Mai	11/02/2003	4.6	6.2	6.3
15	SV016	Dang Lan	25/02/2004	4.3	3.4	6.5
16	SV017	Le Hai	02/01/2003	4.4	3.3	5.1
17	SV018	Bui Linh	11/05/2000	4.4	3.1	5.5
18	SV019	Bui Mai	18/05/2005	3.6	5.2	6.5
19	SV020	Hoang Hai	11/02/2001	4.3	4.2	8.1

Image 6 Create dataset

3.2 Define predicates

I defined the following predicate functions for evaluating student performance:

- **is_passing(student)**: Returns True if the student has scores of 5 or higher in all subjects.
- **is_high_math(student)**: Returns True if the Math score is 9 or above.
- **is_struggling(student)**: Returns True if both Math and CS scores are below 6.
- **improved_in_cs(student)**: Returns True if the CS score is higher than the Math score.

3.3 Evaluating Quantified Statements

I formulated and evaluated the following quantified statements base on the dataset:

Type of quantified	Statement	Truth Value
Universal quantifications	All students passed	False
	All students passed	True
Existential quantifications	There exists a student whose math score is above 9	True
	There is a student whose cs score is higher than his math score.	True
2 Combined/nested statements	For every student, there exists a subject in which they score above 6	False
	For every student who scores below 6 in math, there exists one subject in which they score above 6	False

Table 10 Evaluating Quantified Statements

3.4 Result of Negated Statements

Original Statement	Negation	Truth Value	Meaning
There exists a student who scored above 9 in Math	No student scored above 9 in Math	False	All students have Math scores less than or equal to 9.
There exists a student who improved in CS over Math	No student improved in CS over Math	False	All students have CS scores less than or equal to their Math scores.
For every student, there exists a subject in which they scored above 6	There exists at least one student who scored 6 or lower in all subjects	True	At least one student has scores of 6 or lower in all subjects.
For every student scoring below 6 in Math, there exists a subject where they scored >6	There exists at least one student with Math < 6 and no subject with score > 6	True	At least one student scored below 6 in Math and also scored 6 or lower in all other subjects.

Table 11 Result of Negated Statements

CHAPTER 4. RSA CRYPTOSYSTEM

4.1 Implementation

- **generate_keys(key_size=2048):** Generates a new RSA key pair. The function creates a private key of the specified bit length and derives its corresponding public key. Returns (public_key, private_key).
- **rsa_encrypt(message: bytes, public_key) → bytes:** Encrypts the given plaintext message using the provided public_key. Internally, it initializes a PKCS1_OAEP cipher with the public key and returns the ciphertext bytes.
- **rsa_decrypt(ciphertext: bytes, private_key) → bytes:** Decrypts the given ciphertext using the provided private_key. A PKCS1_OAEP cipher is created with the private key to reverse the encryption and return the original plaintext bytes.
- **measure_timing(message: bytes, public_key, private_key) → tuple:** Measures and returns the time taken for encryption and decryption (in milliseconds) along with the resulting ciphertext and recovered plaintext. It uses time.perf_counter() to record precise timestamps and asserts that decryption matches the original message.
- **display_results(message: bytes, timing_data: tuple):** Prints the original message, raw ciphertext, decrypted message, and the encryption/decryption durations (in ms) to the console, formatted for readability.
- **print_summary_table(results: list):** Takes a list of result tuples (index, length, enc_time, dec_time, success_flag) and prints an ASCII-formatted summary table showing message lengths, timings (in seconds), and success status for each test case.
- **plot_timings(lengths: list, enc_times: list, dec_times: list):** Uses matplotlib to plot encryption and decryption times against message lengths. The chart visualizes how RSA performance varies with the size of the plaintext.

4.2 Test

```
Original message:    Hello, I'm Truong
Encrypted message:   b'lO\x14o\xee\x80j\xf1\xfa\x1d\xbds3\x16EZ\xd4\xb8\xa0\xce\xdam\xaa\x07Ijt\xc
89.\xa8iF}Gdk\xdbs\x8c\xe9\xae\xf2\xe2\xca\xccAo\x91\xe0\xb2\x8d\x10_\x9a\xa3\xc5\xd2\x02\xc47\x05
\x01\xe0\xd70\x9d]\xf7\x03\x05\x11d\xc0d\x00\xdb\xa6\xdf\x82\xc2\x87J\x13\xa2\xe2\x03\xd2*\xaelp\x
\xee\t\xa2\x94O\xdf\xb7\xd4\x08\x93xk\x1b\xc5\xf6\xfc\xc6\xf1\x03\xf0BE\x9e\xe0\xfb\xc76\x13\xd7\r
Decrypted message:   Hello, I'm Truong
Encryption time:     1.775 ms
Decryption time:     8.554 ms
================================================================================
Original message:    My partner is Vu
Encrypted message:   b"\x92WF0\xd7\x13\xc2\xd3Hdha\x90Ci1\x8a\x0c\xd7q\x86\xfbE6\x148\xee\xad\x7f2
8c+!\x8bJYR\xdcJ\x17rV\xcd\xd9\\}A\xc0\xe2mYWL\xfd\xc6[ Vx&\xb5\xb7!M\x02\xb7]\xef1:\x03\xc6\xc2\x
mf\xc1QIML\xa2.;\xdf\xe9\x00\xf1=u,\xae\xcf\x8e\x8d\xa5=\x9b\xc6\xadk\xa0\xf6\xf9\xcfw\x18\xef\xdc
\x18\xef\xdcB)u\xb8rr\xc3\xa8P\x9cV\x927\xfcn\xca\x0b\x7f\xd9\xbc\xbet\xdaI\xa3\xc4D\n4\x95\xaf\x1
\x9c\xf5<\x0ch\xfdD?,\xe8\xf1\x7f\x15Zg\xe0\x19v8\xe9\xdd\x0f\xf9\xd0\x91LN\xb7\x9f@\x
aem,\x1e1\xa9\xd23X=*\xa0\xb44\x95\xaf\x19"
Decrypted message:   My partner is Vu
Encryption time:     1.253 ms
Decryption time:     8.509 ms
================================================================================
Original message:    Thank you for reading our essay
Encrypted message:   b'\x92&\xd4A\xf7oIg\xf3\xf4\xf2\xb4e=\x03q\x86\xbeY&V\x88.cN\xb
dc\xcc\xcd3\x91\x1a\x90v\x95\xb5\xce\xac\xdd\xcdV\xdfZ9\x1f\xcb>_\xac\x85\x0e\x9cO2\
\xef\xb2\xd1\xdc\xeeM\x19\xf0\xaf5@pX\xaa\xbd\x1eg\x10Q\xaf\x01\xb2\xb4\xba6\xce:\x0
04\x99n\x90p\x98\xc9dq\x06\x02\x9d\xfb\x84\x16\xfc\xf2\x8d\x94\x90f\xdb\x0cl\xdc\xf1
8\x0eV\xbcp\xedD\x06\x00\x9c\xfd\xd7%-HD\x04\x989\xecDj\x18\xecW\xa4\x0e\x7f\x98\xaf
\xe8W\x88\x03\x1a\x1d\x97\xf8\x926\x97\xed\x860Ix:\x08\xb4\x8b\xde\xf7\xffO\x00\x89\
x84\xd17\xee\x08.bRb\xe1\xaeZ\xbbw\xba\xa5n\x87\x05\xd2\tx\xcd\xb2X\xfb\xb7\xfb\x9f1
\xe0\x18OqC\x84\xf5\xc0\xc3_\xbcO\xd1\xc2\xa1\xc0\x824\xa3Bv\x9d\xa3\xf9Wz\x17\t\xd3
\xef8o\x05c\x0e\x19\xcc\xd5\x03 o[",*G\x1a\x98\xfa\xaa_\xc4n\xfeG\x84\x1e'
Decrypted message:   Thank you for reading our essay
Encryption time:     1.355 ms
Decryption time:     6.670 ms
================================================================================
```

Image 7 RSA cryptosystem test

4.3 Performance Measurement

4.3.1 Summary of measures

=== SUMMARY TABLE ===

No.	Message Length	Encryption Time (s)	Decryption Time (s)	Result
1	2	0.001643	0.008134	Success
2	8	0.001014	0.007963	Success
3	23	0.001601	0.007127	Success

Image 8 Summary of measures

4.3.2 Graph

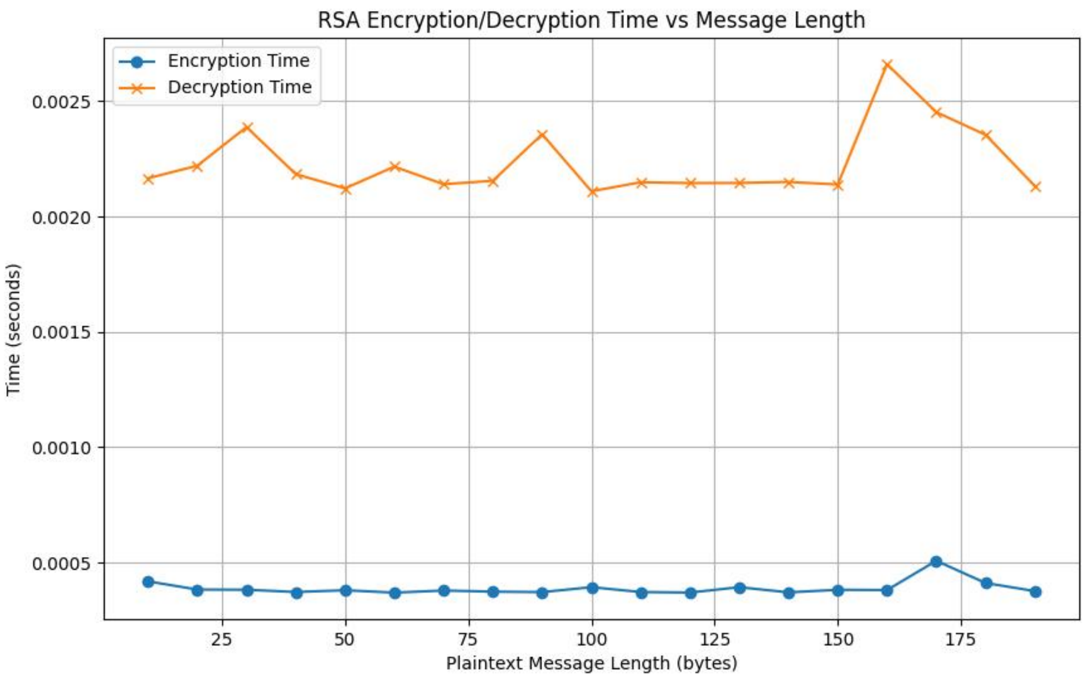

Image 9 Graph

4.4 Discussion

From the measured results and the chart, we observe that RSA decryption takes significantly longer than encryption, and its duration increases rapidly as the message length grows.

Discussing the Limitations of the RSA Cryptosystem:

- Poor Performance on Large Messages: RSA encryption and decryption become very slow when handling large messages, making it unsuitable for encrypting sizable data blocks.
- Message Size Restriction: RSA cannot encrypt messages longer than the key size. For example, a 2048-bit key can encrypt at most 256 bytes; any longer plaintext triggers a "ValueError: Plaintext is too long."
- Asymmetric Overhead: Because RSA is an asymmetric algorithm, both encryption and decryption times grow with message length. For large messages, RSA is not an efficient choice.

4.5 Recommendations

The proposed solution is use a Hybrid Cryptosystem:

- RSA is used only to encrypt the session key for a faster algorithm like AES.
- Then, AES is used to encrypt the large data.

This method is widely used in practice (e.g., HTTPS, VPN, etc.). By applying the Hybrid Cryptosystem approach, we can encrypt significantly larger amounts of data.

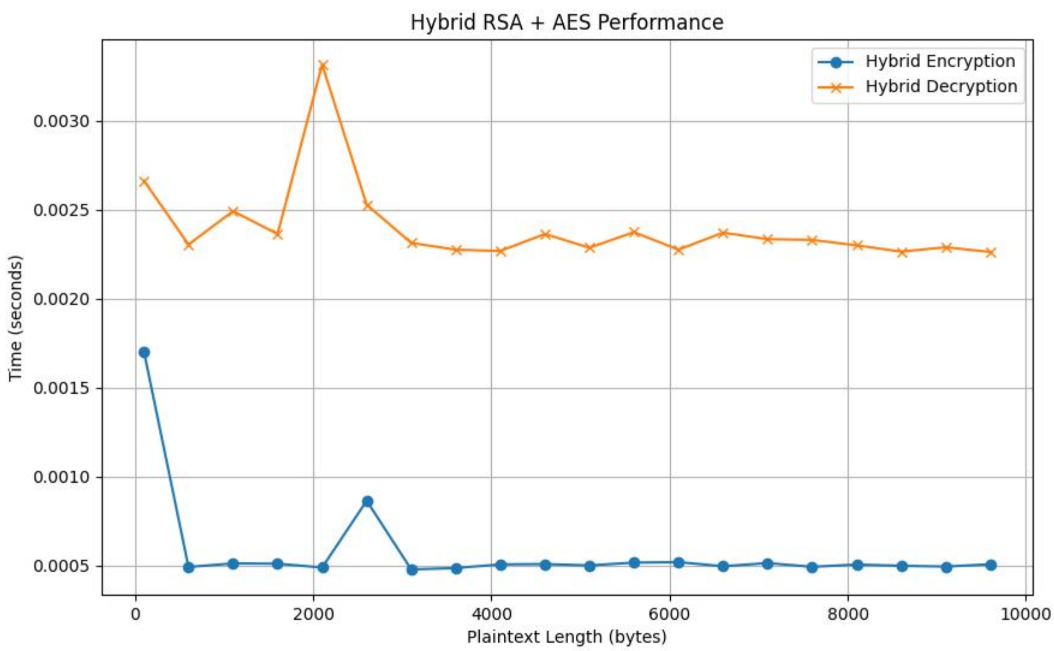

Image 10 Hybrid RSA + AES Performance Graph

Conclusion:

- The RSA cryptosystem is suitable for encrypting short messages or symmetric keys, but it is not appropriate for encrypting large volumes of data.
- The Hybrid Cryptosystem is a simple and effective approach to applying RSA in real-world scenarios.

REFERENCES

[1]. Susanna S. Epp, [2011], Discrete Mathematics with applications, 4th edition, Brooks/Cole, Boston.

[2]. Kenneth H. Rosen, [2012], Discrete mathematics and its applications, 7th edition, McGraw-Hill, New York.

[3]. Allan Stavely, [2014], Programming and mathematical thinking: A gentle introduction to discrete math featuring Python, The New Mexico Tech Press, Socorro, New Mexico.

[4]. R. P. Grimaldi, [2004], Discrete and Combinatorial Mathematics: An Applied Introduction, 5th edition, Pearson Education, Boston.

[5]. Rowan Garnier, John Taylor, [2002], Discrete Mathematics for New Technology, 2nd edition, Institute of Physics, Bristol, Philadelphia.