Here's a **complete explanation** of the **theory**, **algorithm**, and **pseudocode** for writing a **Lex program** that counts **vowels**, **consonants**, and **digits** in the user input.

□ Theory

Lex (Lexical Analyzer) is a tool used to generate a lexical analyzer (tokenizer) — a program that converts a sequence of characters into tokens.
Each pattern (regular expression) in Lex corresponds to an action (usually C code) that executes when the pattern is matched.

Here, we will:

- Take an input string from the user.
- Match patterns for vowels ([AEIOUaeiou]), consonants ([A-Za-z] excluding vowels), and digits ([0-9]).
- Count how many of each occur in the input.

Lex will automatically read the input from **stdin** and match patterns until the end of file (EOF).

□ Algorithm

1. **Start** the program.
2. **Initialize counters**:
   vowel = 0, consonant = 0, digit = 0.
3. **Read input** from the user.
4. **For each character** in the input:
   - If it matches [AEIOUaeiou], increment vowel.
   - Else if it matches [0-9], increment digit.
   - Else if it matches [A-Za-z] but not a vowel, increment consonant.
   - Ignore other characters (spaces, symbols, etc.).
5. **At the end**, print the counts of vowels, consonants, and digits.
6. **Stop.**

💻 Pseudocode

```
BEGIN
   Initialize vowel = 0, consonant = 0, digit = 0
   Read input from user
   For each character ch in input:
      IF ch in [AEIOUaeiou]
         vowel++
      ELSE IF ch in [0-9]
         digit++
      ELSE IF ch in [A-Za-z]
         consonant++
      ENDIF
   END FOR
   Print "Number of vowels = ", vowel
   Print "Number of consonants = ", consonant
   Print "Number of digits = ", digit
END
```

⚙ **Lex Program (.l file)**

```
%{
#include <stdio.h>
int vowel = 0, consonant = 0, digit = 0;
%}

%%
[AEIOUaeiou]      { vowel++; }
[0-9]             { digit++; }
[A-Za-z]          { consonant++; }
[\n]              { /* ignore newline */ }
```

```
.               { /* ignore other characters */ }
%%


int main()
{
   printf("Enter input: ");
   yylex();  // start lexical analysis
   printf("\nNumber of vowels: %d", vowel);
   printf("\nNumber of consonants: %d", consonant);
   printf("\nNumber of digits: %d\n", digit);
   return 0;
}
```

PRACTICAL 2

A Lex program matches input text against regular-expression patterns. When a pattern matches, its action (C code) runs. We define patterns for identifiers/keywords, comments, whitespace and newline characters. We keep counters and increment them in the appropriate actions. For block comments we also scan the matched text to count embedded newlines so the overall line count stays correct.

Algorithm

1. Initialize counters: keywords = 0, identifiers = 0, comments = 0, words = 0, spaces = 0, lines = 0.
2. Define a list of keywords.
3. Read input (stdin) until EOF.
4. For each token matched:
    - If it is a block comment /* ... */: comments++; count \n inside it and add to lines.
    - Else if it is a single-line comment //...: comments++.

- Else if it is an identifier pattern [A-Za-z_][A-Za-z0-9_]*:
  - words++.
  - If token matches a keyword -> keywords++ else identifiers++.
- Else if it is a space ' ' -> spaces++.
- Else if it is newline '\n' -> lines++.
- Ignore other characters (or add more rules if you want to count punctuation, numbers, tabs etc.).

5. After EOF, print the counters.

Pseudocode

```
BEGIN
 keywords = 0
 identifiers = 0
 comments = 0
 words = 0
 spaces = 0
 lines = 0

 load keyword_list

 while (get next token)
  if token matches block_comment (/* ... */)
   comments++
   lines += count_newlines_in(token)
  else if token matches single_line_comment (// ...)
   comments++
  else if token matches identifier_regex
   words++
```

```
    if token in keyword_list
      keywords++
    else
      identifiers++
  else if token is ' '
    spaces++
  else if token is '\n'
    lines++
  end if
 end while

 print counters
END
```

Practical 3

Theory

A Lex program matches input text against regular-expression rules. We define a rule that matches a "word" (a sequence of letters) and in the action check whether its first character is A or a. If it is, we increment a counter. At EOF we print the result.

Algorithm

1. Initialize count = 0.
2. Read input from stdin until EOF.
3. For each token that matches a word (e.g. [A-Za-z]+):
    - If the first character is A or a, increment count.
4. After EOF, print count.
5. End.

Pseudocode

```
BEGIN
 count = 0
 while (there is more input)
  read next word token (letters only)
  if token[0] == 'A' or token[0] == 'a'
   count = count + 1
  end if
 end while
 print "Words starting with A/a:", count
END
```

Lex program (count_A_words.l)

Save this exact text as count_A_words.l.

```
%{
#include <stdio.h>
#include <string.h>

int a_count = 0;
%}

/* Rules section */
%%
[A-Za-z]+    {
            /* yytext is the matched word (letters only) */
            if (yytext[0] == 'A' || yytext[0] == 'a') {
               a_count++;
            }
          }
```

```
/\n/        { /* count newlines if you want to track lines - optional */ }

/[ \t]+/    { /* ignore spaces/tabs */ }

/.          { /* ignore other characters */ }
%%

int main(void)
{
   printf("Enter text (Ctrl+D to finish on Linux/macOS, Ctrl+Z then Enter on Windows):\n");
   yylex(); /* perform scanning */
   printf("\nNumber of words starting with 'A' or 'a' = %d\n", a_count);
   return 0;
}
```

Practical 4

Lex works by matching patterns and executing actions for each match.
We can use regular expressions to identify **uppercase** and **lowercase** letters.
When matched:

- For lowercase letters, we can convert to uppercase using the C library function toupper().
- For uppercase letters, we can convert to lowercase using tolower().

For **option 3**, we'll convert *every character* to lowercase (i.e., "little case" — everything becomes lowercase, including uppercase letters).

⚙ ALGORITHM

1. Start program.
2. Ask the user which conversion they want:
    - 1 → Lowercase to Uppercase
    - 2 → Uppercase to Lowercase
    - 3 → Entire text to Little Case
3. According to the choice:
    - Read input until EOF.
    - Match characters using Lex rules:
        - [a-z] for lowercase
        - [A-Z] for uppercase
        - . for any other character
    - Apply the corresponding transformation using putchar() with toupper() or tolower().
4. Display converted text.
5. Stop.

□ PSEUDOCODE

```
BEGIN
  Print menu options
  Read user choice
  If choice == 1
     For each character in input
        If char is lowercase
           Convert to uppercase
        Else print as it is
  Else if choice == 2
     For each character in input
```

If char is uppercase

Convert to lowercase

Else print as it is

Else if choice == 3

For each character in input

Convert all to lowercase

Print converted text

END

Practical 5

Theory

Hexadecimal (base-16) represents numbers using 16 symbols: 0-9 for values 0–9 and A-F for values 10–15.
To convert a non-negative integer from decimal (base-10) to hexadecimal you repeatedly divide by 16 and collect remainders. The remainders (read in reverse order) give the hex digits.

Key facts:

- Division by 16 extracts the least-significant hex digit (remainder).
- Repeat until the quotient is 0.
- Map remainder 10→A, 11→B, 12→C, 13→D, 14→E, 15→F.
- For negative integers, convert the absolute value and add a leading - (or use two's complement representation if required).
- For fractional parts, multiply the fractional part by 16 repeatedly; the integer parts of the results are the hex fractional digits.

Simple algorithm (integer part) — division-remainder

1. If n == 0 return "0".
2. Initialize an empty stack or string digits.

3. While n > 0:
    - rem = n % 16
    - push rem onto digits (or append to a temporary list)
    - n = n / 16 (integer division)
4. Pop digits and map each 0..15 to 0..9,A..F to form the hex string.
5. If original number was negative, prefix -.

Time complexity: O(log16(n)) = O(log n). Space: O(number of hex digits).

Pseudocode (integer conversion)

```
FUNCTION decimal_to_hex(n: integer) -> string
  IF n == 0
    RETURN "0"
  ENDIF

  negative = false
  IF n < 0
    negative = true
    n = -n
  ENDIF

  digits = empty list

  WHILE n > 0
    rem = n % 16
    prepend rem to digits       // or push to stack
    n = n // 16                 // integer division
  ENDWHILE
```

```
    hex_str = ""
    FOR each d in digits (from first to last)
        IF d < 10
            hex_str = hex_str + char('0' + d)
        ELSE
            hex_str = hex_str + char('A' + (d - 10))
        ENDIF
    ENDFOR

    IF negative
        hex_str = "-" + hex_str
    ENDIF

    RETURN hex_str
END FUNCTION
```

Practical 6

Theory

We treat the input as a sequence of lines. A line “contains .com” if the characters . then c then o then m appear consecutively anywhere in that line.
Lex rules can match whole lines that contain .com (using a pattern that allows any non-newline characters before and after \.com) and separate rules handle other lines and newlines. When we match a .com line we inspect the matched text (yytext) to count how many .com substrings it contains.

Algorithm

1. Initialize counters: total_lines = 0, matched_lines = 0, dotcom_occurrences = 0.
2. Read input until EOF, line by line:

- If a line contains .com:
    - matched_lines++
    - Count occurrences of .com inside the line and add to dotcom_occurrences
    - Print the line (optional)
    - total_lines++
- Else:
    - total_lines++

3. At EOF print total_lines, matched_lines, and dotcom_occurrences.

Time complexity: O(N * L) where N = number of lines and L = average line length (substring counting is linear in line length).

Pseudocode

```
BEGIN
 total_lines = 0
 matched_lines = 0
 dotcom_occurrences = 0

 while (read next line as s)
  total_lines = total_lines + 1
  if ".com" is a substring of s
    matched_lines = matched_lines + 1
    cnt = count_substrings(s, ".com")
    dotcom_occurrences = dotcom_occurrences + cnt
    print "Matched line: " + s
  end if
 end while

 print "Total lines: ", total_lines
```

```
print "Lines containing .com: ", matched_lines
print "Total occurrences of .com: ", dotcom_occurrences
END
```

count_substrings(s, pattern) repeatedly finds pattern in s advancing past the found position to allow overlapping or non-overlapping counting (here .com doesn't self-overlap, so simple advancing after the match is fine).

Practical 7

Postfix Expression (Reverse Polish Notation)

In a **postfix expression**, the **operator** follows its operands.
Example:

5 3 + → (5 + 3)

5 3 2 * + → (5 + (3 * 2))

Why Postfix?

Postfix eliminates the need for parentheses and operator precedence rules because evaluation follows a simple stack-based procedure.

⚙ **ALGORITHM (Evaluation using Stack)**

1. Initialize an empty **stack**.
2. Read the expression **token by token** (left to right).
3. If the token is a **number**, **push** it onto the stack.
4. If the token is an **operator (+, -, *, /)**:
    - **Pop** two operands from the stack: op2 = pop(), op1 = pop()
    - **Apply** the operator: result = op1 <operator> op2
    - **Push** the result back onto the stack.
5. When the expression ends, **the final result** will be the **only value left** on the stack.

□ PSEUDOCODE

```
BEGIN
  create empty stack
  read postfix expression
  for each token in expression:
    if token is number:
      push(token)
    else if token is operator:
      op2 = pop()
      op1 = pop()
      result = op1 operator op2
      push(result)
  end for
  result = pop()
  print result
END
```

Practical 8

Theory (short)

Infix expressions (like 3 + 4 * (2 - 1)) require operator precedence and parentheses handling. Yacc/Bison builds a parser from a grammar and lets you attach semantic actions (C code) that compute expression values (here we use double for numeric values). Lex/Flex tokenizes numbers, operators and parentheses, and passes tokens to the parser.

We use precedence declarations in Yacc:

- * and / have higher precedence than + and -
- unary - is declared with right-associative precedence (%right UMINUS) so -3 * 4 parses as (-3) * 4

Algorithm / Pseudocode

BEGIN

Read a line (infix expression)

Tokenize into numbers and symbols

Parse according to grammar with precedence

For each production evaluate numeric result:

- number -> value
- expr + expr -> sum
- expr * expr -> product
- (expr) -> expr value
- unary minus -> -value

Print result

Repeat until EOF

END

Practical 9

Theory (short)

A for loop header has three *semicolon-separated* parts inside parentheses. A parser (Yacc/Bison) expresses the grammar and checks whether tokens from the scanner (Flex) follow the grammar. We only validate syntax (not semantics): e.g., we won't check types, whether an identifier was declared, etc.

Algorithm

1. Tokenize input into keywords, identifiers, numbers, operators, and punctuation.
2. Parse according to a grammar for for statements:
    - for_stmt -> FOR '(' for_init ';' for_cond ';' for_iter ')' statement
    - for_init -> declaration | expr_list | empty
    - for_cond -> expression | empty

- for_iter -> expr_list | empty

3. If parse succeeds, print Valid for-loop syntax. If a syntax error occurs, yyerror prints the error and parsing continues/aborts as appropriate.

Pseudocode

```
BEGIN
  tokenize input using Lex
  parse tokens using Yacc grammar:
    expect 'for' '('
    parse init (declaration or expression list or empty)
    expect ';'
    parse cond (expression or empty)
    expect ';'
    parse iter (expression list or empty)
    expect ')'
    parse statement (single stmt or compound)
  if parsing reaches end successfully:
    print "Valid for-loop syntax"
  else
    print "Syntax error: ..." with location (approx)
END
```

Practical 10

THEORY

What is Intermediate Code?

Intermediate code (IC) is a low-level representation of a program between **source code** and **machine code**, often used by compilers for optimization and translation.

One common form is **Three-Address Code (TAC)**, where each statement has at most **one operator** and **three operands**:

t1 = b * c

t2 = a + t1

t3 = d / e

t4 = t2 - t3

Goal

Given an arithmetic expression (with +, -, *, /, and parentheses), generate intermediate 3-address code that computes the expression step by step.

⚙ **ALGORITHM**

1. **Lexical Analysis (Lex file)**:
    - Tokenize identifiers, numbers, operators, and parentheses.
    - Pass tokens to Yacc.
2. **Parsing (Yacc file)**:
    - Use grammar rules to ensure correct syntax and operator precedence.
    - For each reduction, generate **temporary variables (t1, t2, …)** and corresponding 3-address code.
3. **Code Generation:**
    - When a production like E → E + E is reduced:
        - Generate a new temporary variable t = E1 + E2
        - Store result in t
    - Print the generated intermediate code in order.
4. **Display** all generated instructions.

□ PSEUDOCODE

Initialize temp_counter = 1

When reducing E -> E1 + E2:

```
    t = newtemp()
    print "t = E1.place + E2.place"
    E.place = t

When reducing E -> E1 - E2:
    t = newtemp()
    print "t = E1.place - E2.place"
    E.place = t

When reducing E -> E1 * E2:
    t = newtemp()
    print "t = E1.place * E2.place"
    E.place = t

When reducing E -> E1 / E2:
    t = newtemp()
    print "t = E1.place / E2.place"
    E.place = t

When reducing E -> (E1):
    E.place = E1.place

When reducing E -> id:
    E.place = id.name
```