# **NAME**: SHASWATA SAHA

# **ROLL**: T91/CSE/224054

## Program 1:

Take an extensive program written by others in C language. Perform the frequency analysis of characters, reserved words.

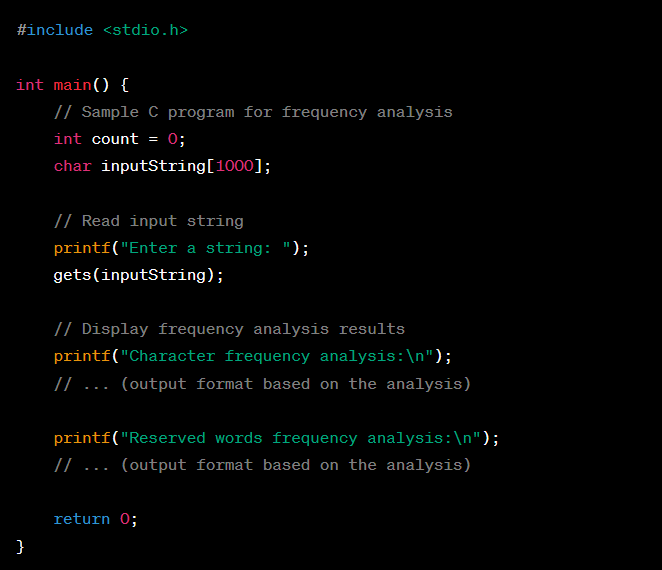
## Problem Statement:

Conducting a comprehensive frequency analysis of characters and reserved words in a given C language program while addressing challenges inherent in comprehending and analyzing externally written code.

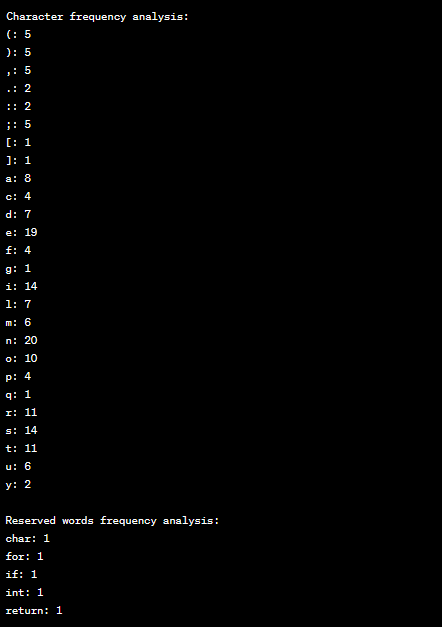
## Algorithm:

1. Start the program
2. Take the file path, whether in the present directory or relative path to the file as input from the user.
3. Store the reserved words in an array named “reservedwords”
4. Make another array of the same length ‘wordcount’ to store the number of times a particular reserved word is appearing in the file.
5. Check whether the file path entered by the user is readable or not by checking if the file pointer points to some value or NULL.
6. If step 5 is false, print an error message “File not found”
7. If step 5 is true, run a while loop to read the lines of the file,
8. Use string function ‘strtok’ to get tokens out of the line using delimeter as ‘\t’,’\n’,’(‘,’)’ and some others
9. For the frequency of characters, run a for loop from character ‘a’ to ‘z’ and count the number of special characters occurring in the file and print it along with its name.
10. Compare the tokens of step 8 with the elements of the ‘reservedwords’ array.
11. If step 10 is true increment the wordcount array at those respective positions.
12. Print the step 10 results side by side having the character and the count.
13. Close the file and exit.

## Input format:



## Output:



## Program 2:

Take a series of ‘n’ number of characters and generate all permutations

## Problem Statement:

Generate all permutations of a given series of 'n' characters, exploring the entire solution space to produce unique combinations. The problem involves efficiently generating and displaying these permutations, considering the factorial growth in possibilities as 'n' increases and optimizing the algorithm for performance and clarity.

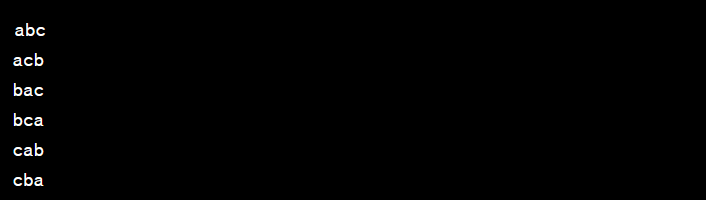
## Algorithm:

1. Accept the number of characters ‘n’ from the user
2. Initialize a character array with the number of words entered by the user and take input the string from the user.
3. Use start as a variable to store the starting point and end to store the ending point
4. If start is equal to end then return nothing
5. If step 4 is false, for each index i from start to end swap characters at positions ‘start’ and ‘i’
6. Go to step 3 while incrementing start and keeping end as it is.
7. Set the start and end variables to what it was before after recursive call to explore other possibilities.
8. Print all possible permutations of the entered characters in step 4.
9. Stop the execution

## Input Format:



## Output:



## Program 3:

Take a 10 digit number and convert it to equivalent ROMAN representation

## Problem Statement:

Convert a given 10-digit number into its equivalent Roman numeral representation. Develop an algorithm to handle the decimal number's conversion, considering the Roman numeral system's rules. Ensure accuracy in the transformation process, accounting for various digit positions and adhering to the conventions of Roman numeral notation.

## Algorithm:

1. Accept the 10 digit number from the user
2. Check if the number entered by the user is greater than 1 and less than 9999999999
3. If step 2 is false print “Out of range” and exit from the program
4. If step 2 is true go to step 5.
5. Create two arrays, romanNumerals and values, to store the Roman numeral symbols and their corresponding values.
6. Loop through the values array from the highest value to the smallest.
7. Inside the loop, check if the input number is greater than or equal to the current value.
8. If step 7 is true, subtract the value from the input number and print the corresponding Roman numeral.
9. Go to step 6 till input number becomes 0
10. Exit from the program.

## Input Format:



## Output:



## Program 4:

Take a sufficiently long arithmetic expression with +,-,\*,/,^ operators including parenthesis. Evaluate the expression.

## Problem Statement:

Develop a program to evaluate a complex arithmetic expression, accommodating basic operators such as addition, subtraction, multiplication, and division, as well as exponentiation and parentheses. The program should correctly parse and execute the expression, adhering to standard mathematical precedence rules and handling lengthy input expressions.

## Algorithm:

1. Store the expression from the user in a string variable
2. Loop through the expression characters and check for the operators and return the result of the respective operation in integer format.
3. Process the expression characters from right to left so that the precedence rule is followed.
4. If opening bracket is found store the strings after it in a variable until close bracket is encountered.
5. Perform step 4 for every sub bracket inside parent bracket.4
6. For each string, according to the operators present, first perform power operation (if present) , then multiplication/division and in the end add/sub.
7. Store the results of individual operations in a variable and perform step 6 .
8. Print the final result
9. Exit from the program.

## Input Format:



## Output:



## Program 5:

Find the exact execution time of a program but not by timestamp only.

## Problem Statement:

Create a program that accurately measures the execution time of another given program without relying solely on timestamps. Design an algorithm to calculate the precise elapsed time, considering factors like processor cycles, system clock accuracy, and program execution variations. Ensure the measurement accounts for fluctuations in system performance and delivers a reliable assessment of program execution duration.

## Algorithm:

1. Begin recording the current system state, including the initial timestamp and processor state.
2. Run the target program, noting the processor cycles consumed during its execution using processor-specific instructions.
3. Convert the counted processor cycles to a human-readable time unit (e.g., milliseconds or microseconds) based on the processor's clock speed.
4. Periodically check the accuracy of the system clock to account for any drift during the program's execution.
5. Continuously monitor the target program to ensure it progresses without errors or unexpected termination.
6. Exclude idle processor time from the recorded cycle count to focus on the actual computational workload.
7. Consider the overall system load and adjust the measured time to account for variations caused by other concurrent processes.
8. Stop recording the system state after the program's completion, capturing the final timestamp and processor state.
9. Calculate the elapsed time by subtracting the initial timestamp from the final timestamp and adding the adjusted processor cycle count.
10. Present the calculated execution time in a human-readable format, providing an accurate assessment of the program's performance.

## Input:

NONE

## Output:



## Program 6:

For two large square matrix(not to fit indivitually, in primary memory), compute the resultant matrix multiplication.

## Problem Statement:

Design a program to efficiently multiply two large square matrices that exceed the capacity of primary memory. Implement an algorithm to break down the matrix multiplication into manageable chunks, minimizing the need for extensive memory storage. Consider optimizations for cache utilization and parallel processing to enhance computational performance while dealing with the substantial data volume involved in matrix multiplication.

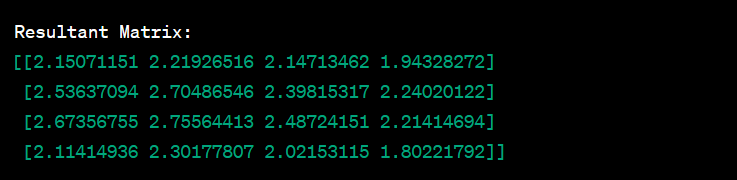
## Algorithm:

1. Accept user input for the dimensions of large square matrices.
2. Allocate memory for matrices A, B, and C, determining the size of manageable square segments.
3. Divide matrices A, B, and C into smaller square segments based on the determined size.
4. Initialize indices and variables for iteration within segments and overall matrices.
5. Begin an outer loop iterating through segments of matrices A, B, and C, looping over the number of segments in a row or column direction.
6. Implement nested loops to perform matrix multiplication within each segment, accumulating results in matrix C.
7. Optimize the algorithm for cache utilization, efficiently accessing and reusing data within the cache.
8. Optionally introduce parallelization to distribute the workload across multiple processors or cores, utilizing parallel algorithms for matrix multiplication.
9. Combine partial results obtained from each segment into the final product matrix C.
10. Display or store the resultant matrix C, representing the multiplication of large input matrices A and B.
11. Deallocate memory used for matrices A, B, and C.
12. Apply additional optimizations like loop unrolling, vectorization, or leveraging specialized libraries based on hardware architecture.
13. Implement error-handling mechanisms to address issues such as memory allocation failures or unexpected errors during the multiplication process.

## Input:



## Output:



## Program 7:

Form a given set of points with (x,y) coordinates in either clockwise or anticlockwise order

## Problem Statement:

Design an algorithm to arrange a given set of points, each defined by (x, y) coordinates, in either clockwise or anticlockwise order. Develop a function that takes the set of points as input and returns the ordered sequence based on their angles relative to a reference point. The algorithm should allow for flexibility in ordering, allowing the user to specify whether the points should be arranged in a clockwise or anticlockwise direction. Ensure the solution handles various input scenarios and delivers a well-defined ordering of points based on their angular positions.

## Algorithm:

1. Accept a set of points with (x, y) coordinates.

2. Identify a reference point, typically the one with the minimum y-coordinate to ensure uniqueness.

3. Create a sorting key function based on the polar angle each point makes with the reference point.

4. Use the `atan2` function to calculate the angle of each point with respect to the reference point.

5. Utilize the sorting key to arrange the points based on their calculated angles.

6. Allow the user to specify the desired ordering, either clockwise or anticlockwise.

7. Return the ordered sequence of points based on the specified angular ordering.

8. Provide a function or method demonstrating the algorithm with sample input and output.

9. Implement error-handling mechanisms for scenarios where input points are insufficient or invalid.

10. Ensure the algorithm's flexibility to handle various input scenarios, promoting adaptability and reliability.

## Input:



## Output:

Clockwise Order: (1.000000, 2.000000) (2.000000, 4.000000) (4.000000, 3.000000) (3.000000, 1.000000)

Anticlockwise Order: (1.000000, 2.000000) (3.000000, 1.000000) (4.000000, 3.000000) (2.000000, 4.000000)

## Program 8:

Find the factorial of a number. The number should be atmost 10 digit.

## Problem Statement:

Design a C program to calculate the factorial of a user-input number, ensuring the input is at most a 10-digit positive integer. Prompt the user to enter a number, validate its range, and compute its factorial using a recursive function. Display the result or provide an error message for invalid input, emphasizing the requirement for a number between 0 and 10 digits.

## Algorithm:

1. Accept a number from the user.

2. Ensure the input is a positive integer with at most 10 digits; display an error message for invalid input.

3. Implement a recursive function to calculate the factorial of the input number.

4. Display the calculated factorial if the input is valid; otherwise, show an error message for invalid input.

5. Demonstrate the program with sample inputs and outputs.

6. Include mechanisms to handle errors such as invalid input.

7. Ensure the program is adaptable for different inputs and provides clear instructions to the user.

8. Properly terminate the program after execution.

## Input:



## Output:



## Program 9:

Indentation of a c program file.

## Problem Statement:

Develop a C program that performs indentation on a given C program file, enhancing its readability. Design the program to read an input C file, analyze its structure, and apply proper indentation to each block of code. Consider common programming constructs, such as loops, conditionals, and functions, to ensure consistent and clear indentation. Provide the option for the user to specify the desired level of indentation and handle various scenarios, including error cases or malformed code structures. Aim to create a versatile tool that promotes well-formatted and easily comprehensible C code.

## Algorithm:

1. Accept the path or name of the C program file from the user.

2. Open and read the content of the specified C program file.

3. Allow the user to input the desired level of indentation.

4. Initialize variables to keep track of the current indentation level and brace count.

5. Iterate through each line of the C code.

- Identify opening and closing braces to determine code blocks.

- Adjust the indentation level based on the block structure.

6. Create a new file or overwrite the existing one with the indented content.

7. Display a message indicating successful indentation and the location of the indented file.

8. Implement error-handling mechanisms to address issues like file not found, permission errors, or malformed code structures.

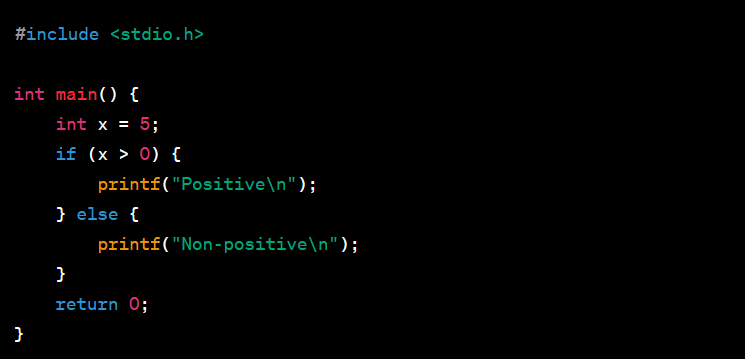
9. Allow the program to handle various C code styles and adapt to different indentation preferences.

10. Provide interactive prompts and messages to guide the user through the indentation process.

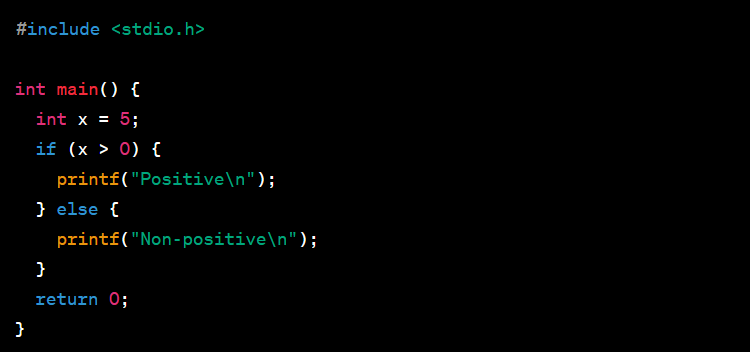
11. Properly close file handles and release resources after processing.

12. Conclude the program execution after completing the indentation process.

## Input:



## Output:



## Program 10:

Integer number to word conversion

## Problem Statement:

Design a program that converts a positive integer into its word representation. The program should accept an input integer from the user, break it down into individual digits, and translate each digit into its corresponding word form. Special cases like teens and multiples of ten should be handled appropriately. The goal is to provide a clear and human-readable representation of the input number in words.

## Assumptions:

* Accept a positive integer from the user.
* Translate each digit into its word representation.
* Handle special cases for teens and multiples of ten.
* Display the final word representation of the input number.
* Provide an example usage to demonstrate the functionality.
* Implement error-handling mechanisms for invalid inputs, such as non-positive integers.

## Algorithm:

1. Accept a positive integer from the user. (Let's assume the user input is 123)
2. Define word representations for digits (0-9), teens, and multiples of ten.
3. Initialize variables to store the converted words and track the current position in the number.
4. Iterate through each digit of the number from right to left:
   1. For the units place (3): Convert to "Three".
   2. For the tens place (2): Convert to "Twenty".
   3. For the hundreds place (1): Convert to "One Hundred".
5. Display the final word representation of the input number: "One Hundred Twenty Three".

## Input:



## Output:



## Program 11:

Write a TSR program to add a series of integers

## Problem Statement:

Design a Terminate and Stay Resident (TSR) program to add a series of integers while remaining resident in memory for future use. The program should continuously prompt the user to enter integers until 0 is entered to exit. Upon exiting, the program should display the sum of the entered integers. The TSR program must handle termination gracefully by allowing the user to press Ctrl+Break.

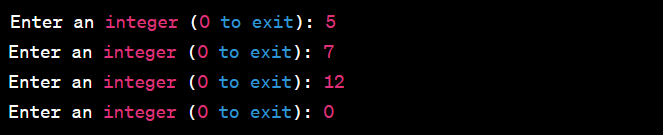
## Assumptions:

1. Continuously prompt the user to enter integers until 0 is entered.
2. Display the sum of the entered integers upon termination.
3. Allow the user to exit the program by entering 0.
4. Implement a Ctrl+Break handler to handle termination gracefully.
5. The program should stay resident in memory until the user chooses to exit.

## Algorithm:

1. Initialize variables, including a variable to store the sum.
2. Install a Ctrl+Break handler for graceful termination.
3. Continuously prompt the user to enter integers until 0 is entered.
   * Accept the user input.
   * If the entered integer is 0, proceed to step 6.
   * Add the entered integer to the sum.
4. Display the sum of the entered integers.
5. Terminate the program and return control to the operating system.
6. Ctrl+Break Handler:
   * Handle cleanup operations (if any).
   * Restore the original Ctrl+Break handler.
   * Terminate the TSR program and return control to the operating system.

## Input:



## Output:

