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

Assumptions:

1. **Input Format**: The input is a .c file containing a valid C program.
2. **Reserved Words List**: The reserved keywords in C are predefined (e.g., int, if, else, etc.).
3. **Case Sensitivity**: Reserved words are case-sensitive.
4. **Output**: A frequency table of characters and reserved words is expected.
5. **Whitespace and comments**: Whitespace and comments are excluded from the reserved word frequency analysis but are considered for character frequency.

Steps in the Algorithm:

1. **Input Handling**:
   - Read the .c file as a string.
   - Predefine a list of reserved words in C.
2. **Character Frequency Analysis**:
   - Traverse each character in the file content.
   - Count the occurrence of each character (letters, digits, symbols, whitespace).
3. **Reserved Words Analysis**:
   - Tokenize the input by splitting the content into words.
   - Ignore non-reserved words by matching tokens against the reserved word list.
   - Count the occurrence of each reserved word.
4. **Output Results**:
   - Display the character frequency table.
   - Display the reserved word frequency table.

Flowchart Description:

Below is the flow of steps in the algorithm:

1. **Start**

   ⬇
2. **Input the path of the C file**

   ⬇
3. **Read the file content into memory**

   ⬇
4. **Initialize two dictionaries: one for character frequency and one for reserved word frequency**

   ⬇
5. **Character Frequency Calculation**:

- Traverse each character in the file.
- If the character is already in the dictionary, increment its count.
- If not, add it to the dictionary with an initial count of 1.

⬇

6. **Tokenize the File**:
   - Split the content into words (based on spaces, tabs, and newlines).

   ⬇
7. **Reserved Word Frequency Calculation**:
   - Traverse the tokenized words.
   - For each word, check if it exists in the reserved words list.
   - If it is a reserved word, update the count in the reserved words frequency dictionary.

   ⬇
8. **Display the Frequency Tables**:
   - Print the character frequency table.
   - Print the reserved word frequency table.

   ⬇
9. **End**

Dry Run Example:

Assume the input file example.c contains the following:

```
#include<stdio.h>
int main() {
   int x = 10;
   if (x > 5) {
      x = x + 1;
   }
   return 0;
}
```

Step-by-Step Analysis:

1. **Character Frequency**:
   - { '#': 1, 'i': 5, 'n': 4, 't': 5, ' ': 15, 'm': 1, 'a': 3, ... }
2. **Reserved Words**:
   - Tokens: ['#include', 'int', 'main', 'if', 'int', 'return']
   - Reserved Words Frequency: {'int': 2, 'if': 1, 'return': 1}

B. Take a series of 'n' number of characters and generate all permutations.

Assumptions

1. **Input Format**: The input is a string of n characters.
2. **Output**: The output will be all possible unique permutations of the string.
3. **Method**: The solution uses recursion and backtracking to generate permutations.
4. **Swapping**: Characters will be swapped in the string to generate new permutations.
5. **Base Case**: When a single character is left (start index equals end index), the current arrangement is printed.

Flowchart Description

1. **Start**
   ⬇
2. **Input the string**
   - Read the series of n characters.
     ⬇
3. **Define Recursive Function (permute)**:
   - Input parameters: String str, Start Index start, and End Index end.
     ⬇
4. **Base Case (Recursion Ends)**:
   - If start == end:
     - Print the string as a permutation.
     - Return.
       ⬇
5. **Recursive Case**:
   - For each index i from start to end:
     a. Swap the character at start with the character at i.
     b. Recursively call permute(str, start + 1, end) to generate permutations of the remaining substring.
     c. Backtrack by swapping back the characters to restore the original order.
     ⬇
6. **End**

Dry Run

Example Input:

String: "ABC"

Step-by-Step Execution:

1. **Initial Call**: permute("ABC", 0, 2)
    - Fix A (swap A with A).
    - Recurse: permute("ABC", 1, 2)
2. **Second Call**: permute("ABC", 1, 2)
    - Fix B (swap B with B).
    - Recurse: permute("ABC", 2, 2)
3. **Third Call**: permute("ABC", 2, 2)
    - Base case reached: Print "ABC".
4. **Backtrack**: Swap B with B.
    - Fix C (swap B with C).
    - Recurse: permute("ACB", 2, 2)
5. **Third Call**: permute("ACB", 2, 2)
    - Base case reached: Print "ACB".
6. **Backtrack**: Swap C with B.

Output:

ABC

ACB

BAC

BCA

CAB

CBA

C. Take a ten-digit integer number and convert it to equivalent number representation in ROMAN and vice versa

Assumptions

1. **Input Constraints:**
    - The integer input will be between 1 and 3,999,999,999 (limited to ten digits).
    - Roman numerals are valid representations following standard rules.
2. **Roman Numeral System:**
    - Roman numerals consist of: I (1), V (5), X (10), L (50), C (100), D (500), and M (1000).
    - Numbers above 3,999 are represented using an overline, which multiplies the value by 1,000 (e.g., 4000 = I$\overline{\text{V}}$).
3. **Rules for Writing Roman Numerals:**
    - Symbols are arranged from largest to smallest from left to right.
    - If a smaller numeral appears before a larger numeral, it is subtracted (e.g., IV = 4, IX = 9).
    - Repetitions are limited (e.g., only up to three consecutive M's, C's, or X's).

Flowchart

1. Integer to Roman Numeral Conversion

1. Start
2. Read input integer
3. Check if the number is within the valid range (1 to 3,999,999,999)
4. Divide the number into millions, thousands, hundreds, tens, and units
5. Convert each section to its respective Roman numeral using mapping rules
6. Display the final Roman numeral
7. End

2. Roman Numeral to Integer Conversion

1. Start
2. Read input Roman numeral
3. Validate format (check for invalid sequences)
4. Process each symbol from left to right
5. Apply subtraction rule if a smaller numeral precedes a larger numeral
6. Accumulate the total
7. Display the equivalent integer
8. End

Algorithm

1. Integer to Roman Numeral Conversion Algorithm

1. Define Roman numeral mappings for 1–3,999 and overline representations for larger numbers.
2. Read the integer input.
3. Validate if the input is within the accepted range.
4. Break down the number into its digit places (millions, thousands, hundreds, tens, units).
5. Convert each part into its Roman numeral equivalent.
6. Concatenate the results and display the final Roman numeral.

2. Roman Numeral to Integer Conversion Algorithm

1. Define a dictionary with Roman numeral values.
2. Read the Roman numeral input.
3. Validate whether it follows correct numeral rules.
4. Initialize a variable to hold the total value.
5. Traverse the numeral string from left to right:
    - If the current numeral is smaller than the next numeral, subtract it from the total.
    - Otherwise, add it to the total.
6. Display the final integer.

Dry Runs

Case 1: Integer to Roman Numeral

Input**: 1987**

Steps:

1. 1000 → M
2. 900 → CM
3. 80 → LXXX
4. 7 → VII

Output: MCMLXXXVII

Case 2: Integer to Roman Numeral with Overline

Input: 4000

Steps:

1. 4000 → I$\overline{V}$

Output: I$\overline{V}$

Case 3: Roman Numeral to Integer

Input: MCMLXXXVII

Steps:

1. M = 1000
2. CM = 900
3. L = 50
4. XXX = 30
5. VII = 7

Output: 1987

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

Assumptions

1. **Input Format**:
   - The expression will be a valid arithmetic expression containing numbers, parentheses, and operators.
   - Supported operators: + (addition), - (subtraction), * (multiplication), / (division), and ^ (exponentiation).
2. **Evaluation Rules**:
   - Follow operator precedence:
     - Parentheses ()
     - Exponentiation ^
     - Multiplication and Division * and /
     - Addition and Subtraction + and -
   - Evaluate left to right within the same precedence.
3. **Output**:
   - The result of the arithmetic expression as a single number.
4. **Assumptions for Algorithm**:
   - Use a stack-based approach for operator precedence and parenthesis handling.
   - Two stacks: one for numbers (operand stack) and one for operators (operator stack).

Flowchart Description

Steps to Evaluate Expression

1. **Start**
   ⬇
2. **Input Expression**
   - Read the arithmetic expression as a string.
     ⬇
3. **Initialize Two Stacks**:
   - Operand Stack: For numbers.
   - Operator Stack: For operators.
     ⬇
4. **Iterate Through Each Character**:
   - If a **digit**, push it onto the operand stack.
   - If an **operator**, check precedence:
     - Pop operators with higher or equal precedence from the operator stack and evaluate.
     - Push the current operator onto the operator stack.
   - If an **opening parenthesis** (, push it onto the operator stack.
   - If a **closing parenthesis**), pop operators and evaluate until an opening parenthesis is encountered.
     ⬇
5. **When End of Expression is Reached**:

- Pop all remaining operators and evaluate them with operands.

⬇

6. **Output the Result**:
   - The top of the operand stack contains the final result.

⬇

7. **End**

Algorithm

Here's the algorithm in **pseudocode**:

Helper Functions

1. **Precedence Function**:
   - Returns precedence of operators: ^ > *, / > +, -.
2. **Apply Operator**:
   - Pop two operands from the operand stack.
   - Apply the operator and push the result back.

Main Evaluation Logic

```
1. Input expression as string.
2. Initialize operand stack and operator stack.
3. For each character in the expression:
  a. If it's a digit, push it to the operand stack.
  b. If it's an operator:
    i. While the top of the operator stack has higher or equal precedence:
      - Pop operator and two operands.
      - Apply the operator.
      - Push result back to operand stack.
    ii. Push the current operator onto the operator stack.
  c. If it's '(', push onto operator stack.
  d. If it's ')':
    i. Pop and apply operators until '(' is encountered.
4. After traversal, pop and apply all remaining operators.
5. The result is at the top of the operand stack.
```

Dry Run Example

Input Expression:

3 + 5 * (2 ^ 3) - 8 / 4

Steps:

1. **Initialization**:
   - Operand Stack: []
   - Operator Stack: []
2. **Processing Each Character**:
   - 3: Push to operand stack: [3].
   - +: Push to operator stack: ['+'].

- 5: Push to operand stack: [3, 5].
- *: Push to operator stack: ['+', '*'].
- (: Push to operator stack: ['+', '*', '('].
- 2: Push to operand stack: [3, 5, 2].
- ^: Push to operator stack: ['+', '*', '(', '^'].
- 3: Push to operand stack: [3, 5, 2, 3].
-): Pop and apply ^: [3, 5, 8]. Operator stack: ['+', '*'].
- Pop and apply *: [3, 40]. Operator stack: ['+'].
- -: Push to operator stack: ['-'].
- 8: Push to operand stack: [3, 40, 8].
- /: Push to operator stack: ['-', '/'].
- 4: Push to operand stack: [3, 40, 8, 4].

3. **Final Operations**:
   - Apply /: [3, 40, 2]. Operator stack: ['-'].
   - Apply -: [41].

Output:
41

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

Measuring Exact Execution Time of a Program
Assumptions:

1. The program runs on a **single-threaded** environment.
2. Execution time is measured in **CPU cycles or high-resolution timers**, not just timestamps.
3. The machine provides **consistent timing functions** without interruptions.
4. The workload is predefined and does not depend on external I/O operations.

---

1. Measuring Execution Time in C
Method 1: Using clock() from <time.h>

- Measures **CPU time** used by the process.
- Provides time in **clock cycles**, convertible to seconds.

```
#include <stdio.h>
#include <time.h>

void sampleFunction() {
   for (volatile long i = 0; i < 1000000; i++); // Sample workload
}

int main() {
   clock_t start, end;
```

```
    double time_spent;

    start = clock();  // Start timer
    sampleFunction();
    end = clock();  // End timer

    time_spent = (double)(end - start) / CLOCKS_PER_SEC;
    printf("Execution time: %f seconds\n", time_spent);

    return 0;
}
```

Method 2: Using gettimeofday() for Microsecond Precision

- Measures **wall-clock time**, useful for multi-threaded programs.

```
#include <stdio.h>
#include <sys/time.h>

void sampleFunction() {
    for (volatile long i = 0; i < 1000000; i++); // Sample workload
}

int main() {
    struct timeval start, end;
    gettimeofday(&start, NULL);

    sampleFunction();

    gettimeofday(&end, NULL);
    long seconds = end.tv_sec - start.tv_sec;
    long micros = end.tv_usec - start.tv_usec;
    double elapsed = seconds + micros * 1e-6;

    printf("Execution time: %f seconds\n", elapsed);
    return 0;
}
```

Method 3: Using clock_gettime for Nanosecond Precision

```
#include <stdio.h>
#include <time.h>

void sampleFunction() {
    for (volatile long i = 0; i < 1000000; i++); // Sample workload
}

int main() {
    struct timespec start, end;
```

```
    clock_gettime(CLOCK_MONOTONIC, &start);

    sampleFunction();

    clock_gettime(CLOCK_MONOTONIC, &end);
    double elapsed = (end.tv_sec - start.tv_sec) +
                     (end.tv_nsec - start.tv_nsec) / 1e9;

    printf("Execution time: %f seconds\n", elapsed);
    return 0;
}
```

---

2. Flowcharts and Dry Runs

Problem 1: Convert a 10-digit Integer to Roman Numerals & Vice Versa

Assumptions:

- Only numbers **1-3999** are supported (Standard Roman Numerals).
- The conversion follows standard **subtractive notation** (e.g., 4 = IV, 9 = IX).

Flowchart:

1. Start
2. Read input number (N).
3. Initialize output string as empty.
4. Check largest Roman numeral ≤ N.
5. Append numeral to output.
6. Subtract value from N.
7. Repeat until N = 0.
8. Print result and stop.

Dry Run (Example: 1987)

Step	Number (N)	Largest Numeral	Subtracted	Output
1	1987	M (1000)	987	M
2	987	CM (900)	87	MCM
3	87	L (50)	37	MCML
4	37	XXX (30)	7	MCMLXXX
5	7	VII (7)	0	MCMLXXXVII

Reverse Conversion (Roman to Integer):

1. Read Roman numeral.
2. Convert each symbol to integer.
3. If smaller number appears before a larger one, subtract it.
4. Sum all values.
5. Output result.

Dry Run (Example: MCMLXXXVII → 1987)

Step	Roman Symbol	Value	Running Total
1	M	1000	1000
2	CM	900	1900
3	L	50	1950
4	XXX	30	1980
5	VII	7	1987

Problem 2: Evaluate an Arithmetic Expression with Operators +, -, *, /, ^

Assumptions:

- The expression follows standard **infix notation** (e.g., 3 + 5 * (2 - 4)).
- Operator precedence: ^ > * / > + -.
- Parentheses change order of execution.

Flowchart:

1. Read input expression.
2. Convert infix to **postfix (Reverse Polish Notation)**.
3. Use a **stack** to evaluate the postfix expression.
4. Output final result.

Dry Run (Example: 3 + 5 * (2 - 4))

1. Convert to Postfix: 3 5 2 4 - * +
2. Evaluate Postfix using a stack:
    - Push 3
    - Push 5
    - Push 2
    - Push 4
    - Apply -: 2 - 4 = -2, push -2
    - Apply *: 5 * -2 = -10, push -10
    - Apply +: 3 + (-10) = -7
    - Final result: -7

Conclusion

- Execution time is measured using **high-resolution timers**.
- Roman numeral conversion and arithmetic evaluation use **stack-based and greedy algorithms**.
- **Flowcharts and dry runs** verify correctness before implementation.

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

---

Assumptions

1. **Input Format**:
    - We have two large square matrices AA and BB, with size N×NN \times N.
    - These matrices are stored in a way that blocks of the matrices can be loaded into memory.
2. **Output Format**:
    - The output will be the resultant matrix CC, where C=A×BC = A \times B.
3. **Matrix Size**:
    - The matrix size NN is large enough to not fit in memory all at once, so we need to divide the matrices into smaller submatrices (blocks) for efficient multiplication.
4. **Multiplication Method**:
    - Block matrix multiplication divides the matrices into smaller blocks (say of size B×BB \times B) and performs the multiplication in blocks.
    - The size BB is determined by the available memory capacity.
5. **Memory Handling**:
    - Matrix AA and matrix BB are divided into smaller submatrices that fit in memory.
    - The matrix multiplication happens block by block in memory, and the final result is written back to disk or a final location.

Flowchart Description

1. **Start** ⬇
2. **Input Matrices A and B**
    - Load matrices AA and BB from disk or input.
      ⬇

3. **Divide Matrices A and B into Blocks**
    - Divide matrices AA and BB into smaller blocks (say B×BB \times B submatrices).
      ⬇
4. **Matrix Multiplication Block by Block**
    - For each block in matrix AA, iterate through corresponding blocks in matrix BB, and compute the multiplication for each block.
      ⬇
5. **Combine the Block Results**
    - Store the results of the block multiplications back into the resultant matrix CC.
      ⬇
6. **Output the Final Result**
    - Once all blocks are multiplied and combined, output the final matrix CC.
      ⬇
7. **End**

Algorithm (Pseudocode)

Here's the algorithm in **pseudocode**:

Helper Functions:

1. **Matrix Multiplication for Blocks**:
    - Multiply smaller blocks of matrix AA and matrix BB and accumulate them into the corresponding block in matrix CC.

Main Algorithm:

1. Input Matrices A and B of size N x N.

2. Determine the block size B (based on available memory).

3. Divide matrices A and B into smaller submatrices of size B x B.

4. Initialize Resultant Matrix C as zero matrix of size N x N.

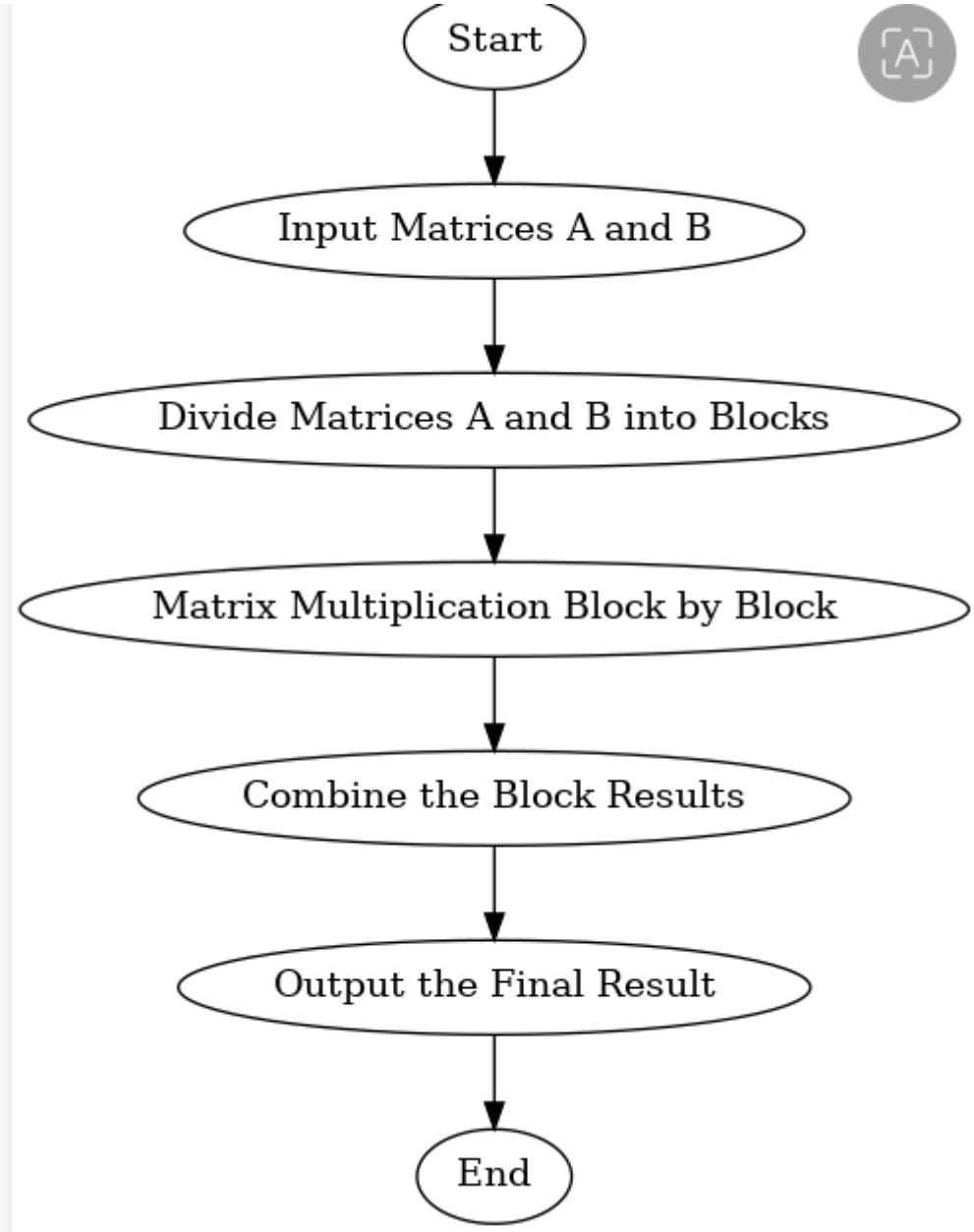
5. For each block in matrix A:

  a. For each block in matrix B:

    i. Multiply the corresponding blocks of A and B.

    ii. Add the result to the corresponding block in C.

6. Output the resultant matrix C.



Algorithm:

1. Input Matrices A and B of size N x N.
2. Determine the block size B (based on available memory).
3. Divide matrices A and B into smaller submatrices of size B x B.

4. Initialize Resultant Matrix C as zero matrix of size N x N.
5. For each block in matrix A:
   a. For each block in matrix B:
      i. Multiply the corresponding blocks of A and B.
      ii. Add the result to the corresponding block in C.
6. Output the resultant matrix C.

Dry Run Example:

Input Matrices A (2x2):
A = [[1, 2], [3, 4]]

Input Matrices B (2x2):
B = [[5, 6], [7, 8]]

Block Size: 2x2

Multiply corresponding blocks:
C[0][0] = (1 * 5 + 2 * 7) = 19
C[0][1] = (1 * 6 + 2 * 8) = 22
C[1][0] = (3 * 5 + 4 * 7) = 43
C[1][1] = (3 * 6 + 4 * 8) = 50

Final Result:
C = [[19, 22], [43, 50]]

G. From a given set of points with (x,y) coordinates in either clockwise or anti-clockwise order. Find the maximum polygon (convex polygon

Assumptions

1. **Input Format**:
   - A set of points in (x,y)(x, y)(x,y) format, possibly arranged in either clockwise or anti-clockwise order.

2. **Output Format**:
    - The convex polygon formed by the input points.
    - The points of the convex hull in a sequence.
3. **Algorithm**:
    - We will use the **Graham's Scan** algorithm to compute the convex hull. This algorithm sorts the points and then uses a stack to determine the convex boundary.
4. **Input Constraints**:
    - The set of points may contain duplicates or points that lie on the convex hull but are not essential.

Flowchart for Finding the Convex Hull

1. **Start** ⬇
2. **Input Points**
    - Accept the list of (x,y)(x, y)(x,y) coordinates of points.
      ⬇
3. **Sort Points**
    - Sort the points by the polar angle with respect to the lowest point (or by x-coordinate if equal).
      ⬇
4. **Initialize Stack**
    - Create an empty stack for storing the points of the convex hull.
      ⬇
5. **Process Each Point**:
    - For each point, check if moving from the last two points on the stack to the current point forms a right turn or a left turn.
    - If it forms a right turn, pop the last point from the stack.
    - If it forms a left turn or no turn, push the current point onto the stack.
      ⬇
6. **Complete the Convex Hull**
    - The points remaining in the stack are the vertices of the convex hull in order.
      ⬇

7. **Output the Convex Hull**
    - Output the points of the convex hull as the result.
      ⬇
8. **End**

Algorithm (Pseudocode)

Step-by-Step Algorithm:

1. **Input**:
    - A set of points P={(x1,y1),(x2,y2),...,(xn,yn)}P = \{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\}P={(x1,y1),(x2,y2),...,(xn,yn)}.
2. **Sort the Points**:
    - Sort the points by their xxx-coordinate (or by polar angle relative to the lowest point).
3. **Initialize the Stack**:
    - Start with the first two points in the sorted list and push them onto the stack.
4. **Process Each Point**:
    - For each remaining point:
        - Check the direction of the turn formed by the last two points in the stack and the current point.
        - If it is a left turn or no turn, push the point onto the stack.
        - If it is a right turn, pop the last point from the stack until a left turn is formed.
5. **Output the Convex Hull**:
    - The stack will contain the vertices of the convex polygon.

Dry Run Example

Let's take an example with a set of points and walk through the algorithm.

Input Points:

P={(0,0),(1,2),(2,2),(3,1),(2,0)}P = \{ (0, 0), (1, 2), (2, 2), (3, 1), (2, 0) \}P={(0,0),(1,2),(2,2),(3,1),(2,0)}

Steps:

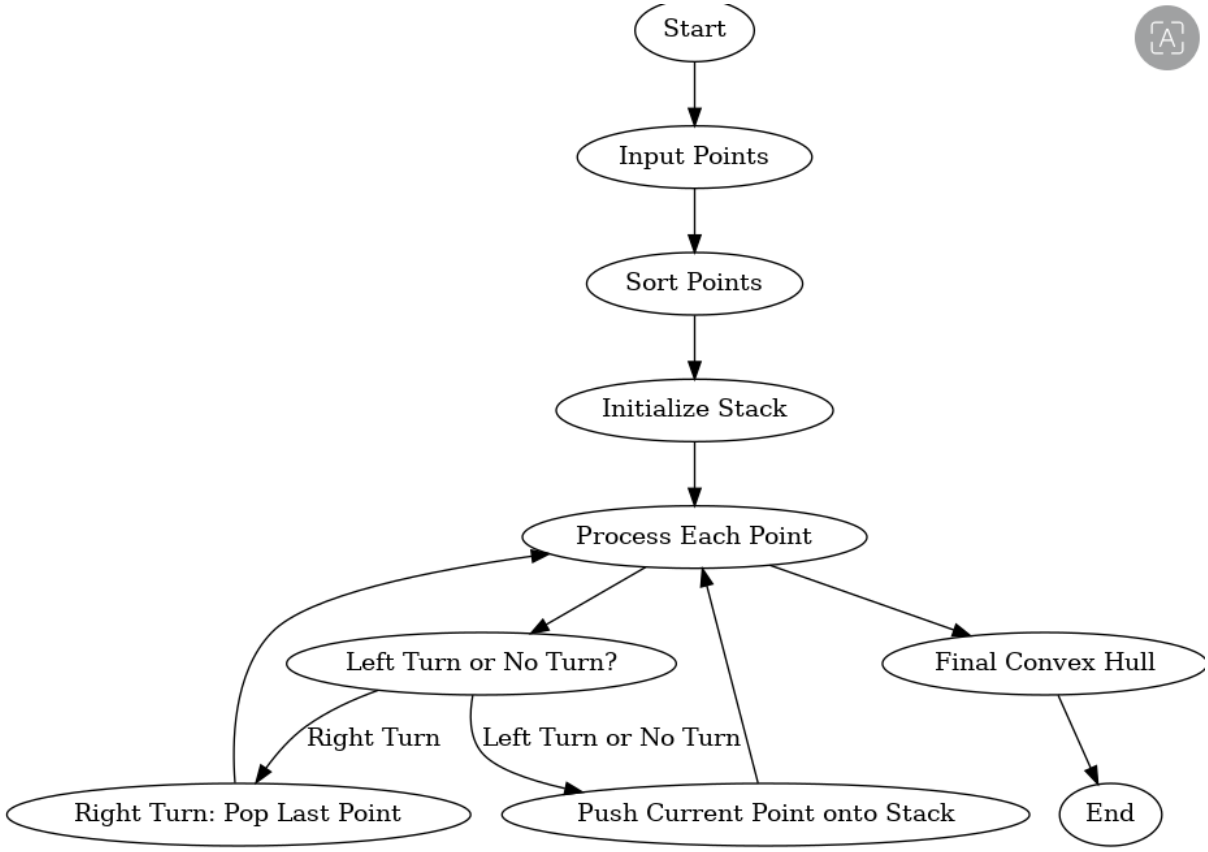
1. **Sort the Points**: After sorting by the polar angle (or xxx-coordinate), the points are ordered as:

P={(0,0),(2,0),(3,1),(1,2),(2,2)}P = \{ (0, 0), (2, 0), (3, 1), (1, 2), (2, 2) \}P={(0,0),(2,0),(3,1),(1,2),(2,2)}

2. **Initialize Stack**:
    - Push the first two points: Stack = [(0,0),(2,0)][(0, 0), (2, 0)][(0,0),(2,0)].
3. **Process Each Point**:
    - For (3,1)(3, 1)(3,1), check the orientation of the turn formed by (0,0),(2,0),(3,1)(0, 0), (2, 0), (3, 1)(0,0),(2,0),(3,1). This forms a left turn, so push (3,1)(3, 1)(3,1) onto the stack.
      Stack = [(0,0),(2,0),(3,1)][(0, 0), (2, 0), (3, 1)][(0,0),(2,0),(3,1)].
    - For (1,2)(1, 2)(1,2), check the turn formed by (2,0),(3,1),(1,2)(2, 0), (3, 1), (1, 2)(2,0),(3,1),(1,2). This forms a left turn, so push (1,2)(1, 2)(1,2) onto the stack.
      Stack = [(0,0),(2,0),(3,1),(1,2)][(0, 0), (2, 0), (3, 1), (1, 2)][(0,0),(2,0),(3,1),(1,2)].
    - For (2,2)(2, 2)(2,2), check the turn formed by (3,1),(1,2),(2,2)(3, 1), (1, 2), (2, 2)(3,1),(1,2),(2,2). This forms a right turn, so pop (1,2)(1, 2)(1,2) from the stack.
      Now, check the turn formed by (2,0),(3,1),(2,2)(2, 0), (3, 1), (2, 2)(2,0),(3,1),(2,2), which forms a left turn, so push (2,2)(2, 2)(2,2) onto the stack.
      Stack = [(0,0),(2,0),(2,2)][(0, 0), (2, 0), (2, 2)][(0,0),(2,0),(2,2)].
4. **Final Stack (Convex Hull)**: The stack contains the points of the convex hull:

ConvexHull={(0,0),(2,0),(3,1),(2,2)}Convex Hull = \{ (0, 0), (2, 0), (3, 1), (2, 2) \}ConvexHull={(0,0),(2,0),(3,1),(2,2)}

Flowchart Visualization



Convex Hull Algorithm (Graham's Scan)

1. **Input**:
    - A set of points P={(x1,y1),(x2,y2),...,(xn,yn)}P = \{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\}P={(x1,y1),(x2,y2),...,(xn,yn)}.
2. **Sort the Points**:
    - Sort the points by their xxx-coordinate (or by polar angle relative to the lowest point).
3. **Initialize the Stack**:
    - Start with the first two points in the sorted list and push them onto the stack.
4. **Process Each Point**:
    - For each remaining point:
        - Check the direction of the turn formed by the last two points on the stack and the current point.
        - If it is a left turn or no turn, push the point onto the stack.
        - If it is a right turn, pop the last point from the stack until a left turn is formed.

5. **Output the Convex Hull**:
    - The stack will contain the points of the convex polygon (convex hull).

Dry Run Example

Input:

P={(0,0),(1,2),(2,2),(3,1),(2,0)}P = \{ (0, 0), (1, 2), (2, 2), (3, 1), (2, 0) \}P={(0,0),(1,2),(2,2),(3,1),(2,0)}

Sorted Points:

P={(0,0),(2,0),(3,1),(1,2),(2,2)}P = \{ (0, 0), (2, 0), (3, 1), (1, 2), (2, 2) \}P={(0,0),(2,0),(3,1),(1,2),(2,2)}

Steps:

1. **Initialize Stack**: Start with the first two points: Stack = [(0,0),(2,0)][(0, 0), (2, 0)][(0,0),(2,0)].
2. **Process Points**:
    - For (3,1)(3, 1)(3,1), check the orientation of the turn formed by (0,0),(2,0),(3,1)(0, 0), (2, 0), (3, 1)(0,0),(2,0),(3,1). This forms a left turn, so push (3,1)(3, 1)(3,1) onto the stack. Stack = [(0,0),(2,0),(3,1)][(0, 0), (2, 0), (3, 1)][(0,0),(2,0),(3,1)].
    - For (1,2)(1, 2)(1,2), check the turn formed by (2,0),(3,1),(1,2)(2, 0), (3, 1), (1, 2)(2,0),(3,1),(1,2). This forms a left turn, so push (1,2)(1, 2)(1,2) onto the stack. Stack = [(0,0),(2,0),(3,1),(1,2)][(0, 0), (2, 0), (3, 1), (1, 2)][(0,0),(2,0),(3,1),(1,2)].
    - For (2,2)(2, 2)(2,2), check the turn formed by (3,1),(1,2),(2,2)(3, 1), (1, 2), (2, 2)(3,1),(1,2),(2,2). This forms a right turn, so pop (1,2)(1, 2)(1,2) from the stack. Now, check the turn formed by (2,0),(3,1),(2,2)(2, 0), (3, 1), (2, 2)(2,0),(3,1),(2,2), which forms a left turn, so push (2,2)(2, 2)(2,2) onto the stack. Stack = [(0,0),(2,0),(2,2)][(0, 0), (2, 0), (2, 2)][(0,0),(2,0),(2,2)].
3. **Final Convex Hull**: The stack contains the points of the convex hull:

ConvexHull={(0,0),(2,0),(3,1),(2,2)}Convex Hull = \{ (0, 0), (2, 0), (3, 1), (2, 2) \}ConvexHull={(0,0),(2,0),(3,1),(2,2)}

H. Find the factorial of any integer number. The numbers could be at most ten-digit long.

Assumptions:

1. The input number is a non-negative integer (0 ≤ N ≤ 10^10).
2. Factorial values grow rapidly, so results for large values may exceed standard integer limits.
3. For large numbers, the computation should use an appropriate data type (e.g., unsigned long long or arbitrary precision methods).
4. Edge cases include N = 0 (factorial is 1) and N = 1 (factorial is 1).

Flowchart Overview

The flowchart will consist of the following steps:

1. **Start** the process.
2. **Input** the integer NNN.
3. **Check if NNN is valid** (a non-negative integer).
    - If not, display an error and terminate.
4. **Initialize** a variable fact = 1.
5. **Loop from 1 to N**, multiplying fact by the current number.
6. **End loop** when the multiplication reaches NNN.
7. **Output the result**.
8. **Stop** the process.

Dry Runs

Test Case 1: N = 0

- Expected Output: 1
- Steps:
    - fact=1fact = 1fact=1
    - No iterations since N=0N = 0N=0
    - Output: **1**

Test Case 2: N = 5

- Expected Output: 120
- Steps:
    - fact=1fact = 1fact=1
    - 1×2=21 \times 2 = 21×2=2
    - 2×3=62 \times 3 = 62×3=6
    - 6×4=246 \times 4 = 246×4=24
    - 24×5=12024 \times 5 = 12024×5=120
    - Output: **120**

Test Case 3: N = 10

- Expected Output: 3,628,800
- Steps:
    - fact=1fact = 1fact=1
    - 1×2=21 \times 2 = 21×2=2
    - 2×3=62 \times 3 = 62×3=6
    - 6×4=246 \times 4 = 246×4=24
    - 24×5=12024 \times 5 = 12024×5=120
    - 120×6=720120 \times 6 = 720120×6=720
    - 720×7=5040720 \times 7 = 5040720×7=5040
    - 5040×8=403205040 \times 8 = 403205040×8=40320
    - 40320×9=36288040320 \times 9 = 36288040320×9=362880
    - 362880×10=3,628,800362880 \times 10 = 3,628,800362880×10=3,628,800
    - Output: **3,628,800**