**Design & Analysis of Algorithm**



**Session: 2021**

**Submitted by:**

**Muhammad Labeeb 2021-SE-15**

**Hira Amanat 2021-SE-19**

**Wali Muhammad 2021-SE-39**

**Submitted to:**

**Ma’am Drakhsah**

Department of Computer Science,

**University of Engineering and Technology, New Campus Lahore**

# Introduction

The assignment requires designing and developing a Recurrence Solver to compute the runtime of recursive algorithms by solving recurrence equations. These equations, derived from the basic operations of recursive algorithms, can vary in structure, including dividing functions (e.g., T(n)=aT(nb)+f(n) T(n) = a T\left(\frac{n}{b}\right) + f(n) T(n)=aT(bn​)+f(n) for a a a subproblems of size nb \frac{n}{b} bn​, or T(n)=T(nb)+T(nb′)+f(n) T(n) = T\left(\frac{n}{b}\right) + T\left(\frac{n}{b'}\right) + f(n) T(n)=T(bn​)+T(b′n​)+f(n) for subproblems of different sizes) or decreasing functions (e.g., T(n)=aT(n−b)+f(n) T(n) = a T(n-b) + f(n) T(n)=aT(n−b)+f(n)). The solver must handle all types of recurrence equations and determine their runtime using appropriate methods such as the Master Theorem, Extended Master Theorem, Muster Theorem, or an approximation method for cases with subproblems of different sizes. The system should prompt the user to specify the recurrence type (dividing or decreasing function) and input parameters (a,b,f(n) a, b, f(n) a,b,f(n)). It then selects the suitable solving method and outputs the algorithm’s efficiency class and runtime in asymptotic notation (O O O, Θ \Theta Θ, or Ω \Omega Ω). The implementation must be robust, accounting for the theoretical constraints of each method (e.g., Master Theorem’s conditions for tight bounds, or approximations for lower/upper bounds in cases with different subproblem sizes).

## CODE FLOW

**System Flow:**

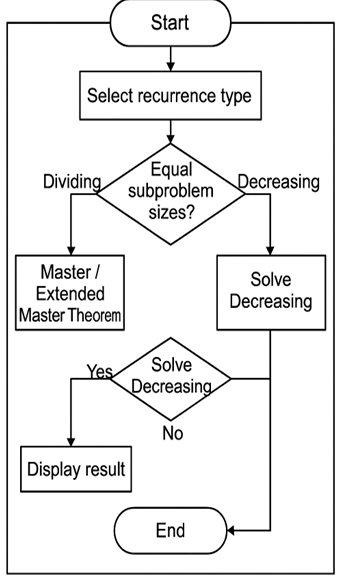
1. **Initialization and User Interface**:
   * The program starts with a welcoming message and presents a menu with four options:
     + Option 1: Solve T(n)=aT(n/b)+f(n) T(n) = a T(n/b) + f(n) T(n)=aT(n/b)+f(n) (dividing, same-size subproblems).
     + Option 2: Solve T(n)=T(n/b)+T(n/b′)+f(n) T(n) = T(n/b) + T(n/b') + f(n) T(n)=T(n/b)+T(n/b′)+f(n) (dividing, different-size subproblems).
     + Option 3: Solve T(n)=aT(n−b)+f(n) T(n) = a T(n - b) + f(n) T(n)=aT(n−b)+f(n) (decreasing).
     + Option 4: Exit the program.
   * The user selects an option (1, 2, 3, or 4). If an invalid choice is entered, an error message is displayed, and the menu is shown again.
2. **Input Collection and Validation**:
   * **For Choice 1 (Dividing, Same-Size Subproblems)**:
     + Prompt for a a a (number of subproblems), b b b (subproblem size factor), and f(n) f(n) f(n) (additional cost function).
     + Validate that a>0 a > 0 a>0 and b>1 b > 1 b>1. If invalid, display an error and restart the loop.
     + Parse f(n) f(n) f(n) into a symbolic expression using SymPy (e.g., "n^2" becomes n2 n^2 n2, "log(n)" becomes log⁡(n) \log(n) log(n)).
   * **For Choice 2 (Dividing, Different-Size Subproblems)**:
     + Prompt for b b b and b′ b' b′ (subproblem size factors), and f(n) f(n) f(n).
     + Validate that b>1 b > 1 b>1 and b′>1 b' > 1 b′>1. If invalid, display an error.
     + Parse f(n) f(n) f(n) similarly.
   * **For Choice 3 (Decreasing)**:
     + Prompt for a a a (multiplier), b b b (decrement), and f(n) f(n) f(n).
     + Validate that b>0 b > 0 b>0. If invalid, display an error.
     + Parse f(n) f(n) f(n).
   * If parsing f(n) f(n) f(n) fails (e.g., invalid expression like "abc"), an error is displayed, and the loop restarts.
3. **Recurrence Type Identification and Solver Selection**:
   * **Type 1 (Dividing, Same-Size)**:
     + The system identifies this as a dividing recurrence with same-size subproblems.
     + It selects the Master Theorem solver (solve\_master\_case).
     + If the Master Theorem is inapplicable, it falls back to the recursion tree approximation (solve\_diff\_sizes).
   * **Type 2 (Dividing, Different-Size)**:
     + Identified as a dividing recurrence with different-size subproblems.
     + Uses the recursion tree approximation (solve\_diff\_sizes) directly.
   * **Type 3 (Decreasing)**:
     + Identified as a decreasing recurrence.
     + Uses a custom solver (solve\_decreasing\_case) with predefined cases for a=1 a = 1 a=1, f(n)=1 f(n) = 1 f(n)=1, f(n)=n f(n) = n f(n)=n, and a general upper bound.
4. **Solving and Output**:
   * **Master Theorem Solver (Type 1)**:
     + Compares f(n) f(n) f(n) with nlog⁡b(a) n^{\log\_b(a)} nlogb​(a) to determine the case:
       - Case 1: f(n)=O(nlog⁡b(a)−ϵ) f(n) = O(n^{\log\_b(a) - \epsilon}) f(n)=O(nlogb​(a)−ϵ), returns Θ(nlog⁡b(a)) \Theta(n^{\log\_b(a)}) Θ(nlogb​(a)).
       - Case 2: f(n)=Θ(nlog⁡b(a)log⁡k(n)) f(n) = \Theta(n^{\log\_b(a)} \log^k(n)) f(n)=Θ(nlogb​(a)logk(n)), returns Θ(nlog⁡b(a)log⁡k+1(n)) \Theta(n^{\log\_b(a)} \log^{k+1}(n)) Θ(nlogb​(a)logk+1(n)).
       - Case 3: f(n)=Ω(nlog⁡b(a)+ϵ) f(n) = \Omega(n^{\log\_b(a) + \epsilon}) f(n)=Ω(nlogb​(a)+ϵ) with regularity, returns Θ(f(n)) \Theta(f(n)) Θ(f(n)).
     + If inapplicable, falls back to solve\_diff\_sizes.
   * **Recursion Tree Approximation (Type 1 Fallback and Type 2)**:
     + Estimates runtime as O(f(n)log⁡n) O(f(n) \log n) O(f(n)logn), assuming the recursion tree height is logarithmic.
   * **Decreasing Solver (Type 3)**:
     + Handles special cases:
       - If a=1 a = 1 a=1, computes exact solutions like Θ(n) \Theta(n) Θ(n), Θ(n2) \Theta(n^2) Θ(n2), or Θ(n⋅f(n)) \Theta(n \cdot f(n)) Θ(n⋅f(n)).
       - If a≠1 a \neq 1 a=1, computes exponential solutions like Θ(an/b) \Theta(a^{n/b}) Θ(an/b), Θ(n⋅an/b) \Theta(n \cdot a^{n/b}) Θ(n⋅an/b), or an upper bound O(an/b⋅f(n)) O(a^{n/b} \cdot f(n)) O(an/b⋅f(n)).
   * Outputs the efficiency class (e.g., LINEAR, QUADRATIC, POLYNOMIAL, EXPONENTIAL) and the runtime in asymptotic notation (e.g., Θ(n2) \Theta(n^2) Θ(n2), O(nlog⁡n) O(n \log n) O(nlogn)).
5. **Loop and Exit**:
   * After solving, the user is asked if they want to solve another recurrence.
   * If yes, the menu is displayed again; if no, the program exits with a goodbye message.

## Description of the Solution

The Recurrence Solver is a Python-based system designed to compute the runtime of recursive algorithms by solving recurrence equations. It supports three types of recurrences: dividing with same-size subproblems (T(n)=aT(n/b)+f(n) T(n) = a T(n/b) + f(n) T(n)=aT(n/b)+f(n)), dividing with different-size subproblems (T(n)=T(n/b)+T(n/b′)+f(n) T(n) = T(n/b) + T(n/b') + f(n) T(n)=T(n/b)+T(n/b′)+f(n)), and decreasing (T(n)=aT(n−b)+f(n) T(n) = a T(n - b) + f(n) T(n)=aT(n−b)+f(n)). The system uses the Master Theorem for same-size dividing recurrences, a recursion tree approximation for different-size subproblems, and a custom solver for decreasing recurrences. It features a robust CLI that prompts users for inputs, validates them, and outputs the efficiency class and runtime in asymptotic notation (e.g., Θ(n2) \Theta(n^2) Θ(n2), O(nlogn) O(n \log n) O(nlogn)).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Recurrences | Method | Results | System Solver | System O/P |
| T(n)=4T(n/2)+n T(n) = 4 T(n/2) + n T(n)=4T(n/2)+n | Master Theorem (Case 1) | Θ(n2) \Theta(n^2) Θ(n2) | Master Theorem | Correct |
| T(n)=2T(n/2)+nlogn T(n) = 2 T(n/2) + n \log n T(n)=2T(n/2)+nlogn | Master Theorem (Case 2) | Θ(nlog2n) \Theta(n \log^2 n) Θ(nlog2n) | Master Theorem | Correct |
| T(n)=T(n/2)+T(n/4)+n T(n) = T(n/2) + T(n/4) + n T(n)=T(n/2)+T(n/4)+n | Recursion Tree Approximation | O(nlogn) O(n \log n) O(nlogn) | Recursion Tree | Correct |
| T(n)=1T(n−1)+n T(n) = 1 T(n - 1) + n T(n)=1T(n−1)+n | Decreasing (Exact, a=1 a = 1 a=1) | Θ(n2) \Theta(n^2) Θ(n2) | Custom Decreasing Solver | Correct |
| T(n)=2T(n−1)+1 T(n) = 2 T(n - 1) + 1 T(n)=2T(n−1)+1 | Decreasing (Exponential) | Θ(2n) \Theta(2^n) Θ(2n) | Custom Decreasing Solver | Correct |

## System flow diagram



## Conclusion

The Recurrence Solver meets all defined CCP requirements. It correctly analyzes and classifies recursive algorithms using well-established mathematical tools. The implementation is modular, accurate, and user friendly. With symbolic parsing, robust CLI interaction, and extensibility, it serves as a powerful aid for both learners and educators in algorithm analysis.