Summer 2011

Algorithms Analysis System: Recurrences

Anchit Sharma
San Jose State University

Algorithms Analysis System: Recurrences

A Writing Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of Requirements for the Degree

Master of Science

by

Anchit Sharma

Spring 2011

ABSTRACT

Algorithms which are recursive have running times which can be described by recurrence equations or recurrences. These equations determine the overall running time complexity of the algorithm.

This project intends to create a mechanism for

- auto generating recurrence equations of the form $T(n) = a(T(n)/b) + f(n)$
- creating a computational method for solving them and generating running times i.e. $O(f(n))$ or $\Omega(f(n))$.
- presenting students with a way to verify their manually computed answers with the solution generation by the project
- generating grading and feedback for their solution

The exercises will utilize the 'Substitution Method' and the 'Master Method' logic to compute running time of the algorithms. Also, practice exercises for comparison of various 'Order of Growth Rates' will be provided.

The project will contribute towards the teaching of analysis of algorithms.

ACKNOWLEDGEMENTS

I thank my advisor, Dr. David Taylor, whose guidance and support is priceless. Dr. Taylor is an educator in the truest sense of the word. His continuous pool of suggestions and ideas helped me visualize the project design. I very much appreciate Dr. Jon Pearce's and Dr. Jeff Smith's participation as thesis committee members.

It has been a challenging, yet rewarding journey which I could not have completed alone and am grateful for your support.

Thank you.

Table of Contents

List of Figures

1.0 INTRODUCTION

This section talks about the need for a system to aid algorithm teaching. It talks of how algorithm analysis tools can be beneficial in learning algorithm theory.

1.1 Need for Algorithms Analysis and Automatic Assessment

AA (Automatic assessment) tools have been gradually developed to illustrate core concepts in theoretical sciences. AA tools are now becoming part of university education. The field of computer science can leverage benefits from these tools especially in the teaching of data structures and algorithms which present the core foundation of computer science education. Visualization of manipulation of data structures and algorithms analysis has been undergoing research in the universities around the world. The ITiCSE (Innovation and Technology in Computer Science Education) group which is sponsored by ACM indicates that the use of Algorithms Analysis AA tools brings positive influence in the understanding of core concepts and benefits computer science learning.

1.2 Related Work

There exist automatic assessment tools for teaching data structures through visualization techniques. Most of them have been developed for academic use by universities. One such Algorithms Analysis AA teaching system which has contributed strongly to the field of computer science education is TRAKLA2 – Software Visualization Group from the Department of Computer Science and Engineering, Helsinki University of Technology. TRAKLA2 is an environment for learning data structures and algorithms. The system provides algorithm simulation exercises that can be

automatically graded. The grading is based on comparison between the learner made simulation sequence and a sequence produced by an actual algorithm.

1.3 Problems Addressed

There has been prior research and development of Algorithm Analysis Tools but most of them rely on the concept of visualization of algorithms on data structures. Recurrences and asymptotic analysis is a fairly untouched area in terms of creating a system to aid its teaching and analysis. Creating an interface for effective analysis and teaching of recurrence solving algorithms is the key challenge. The system needs to be able to provide a problem statement, effectively display its solution, and provide an interface for automatic assessment.

There needs to be a computation engine that will be able to simulate the Substitution method and the Master method used for solving recurrence relationships. There has not been any known prior work in making computation schemes for these algorithms and turning them into an algorithm analysis tool.

2.0 Theory

This project provides simulation exercises with the objective of creating a deeper understanding and learning of recurrence relationships and asymptotic analysis of algorithms. The two methods used to determine asymptotic time complexity of recurrence relationships used in this project are the Substitution Method and the Master Theorem. A brief description of these are provided for a quick recap.

2.1 Substitution Method

The substitution method is a way of proving an asymptotic bound on a recurrence by induction. The guessed answer is substituted into the recurrence equation and the inductive hypothesis is applied. The induction will always be of the same basic form, but it is still important to state the property we are trying to prove.

Below is an example of the substitution method in use:

Let the recurrence be $T(n) = 2T(n/2) + n$.

Lets guess its upper bound to be n (lg n), therefore the solution is $T(n) = O(n (\lg n))$ which means that $T(n) <= cn (\lg n)$ for an appropriate choice of the constant $c > 0$.

We start by assuming that this bound holds for n/2 i.e., that $T(n/2) <= c(n/2) (\lg (n/2))$.

Substituting into the recurrence yields

$T(n) <= 2c(n/2) (\lg (n/2)) + n <= cn (\lg n)$ (as stated above)

For the inductive hypothesis to be complete

$2c(n/2) (\lg (n/2)) + n <= cn (\lg n)$

must hold for a large enough c and for some $n >= n_0$ where n_0 is a constant.

After solving it turns out that

$T(n) <= cn (\lg n)$ if $c >= 1$ and $n_0 = 1$

Thus $T(n) = O(n (\lg n))$.

2.2 Master Theorem

The master method provides a methodical way for solving recurrences of the form $T(n) = a(n/b) + f(n)$ where $a > 1$ and $b > 1$ are constants and f(n) is an asymptotically positive function.

The theorem has three cases in which may recurrences fall in. The three cases of the Master Theorem are:

1. If f(n) = $O(n^{\log_b a - e})$ for some constant e > 0, then $T(n) = \Theta(n^{\log_b a})$.

2. If f(n) = $(n^{\log_b a})$, then $T(n) = \Theta\ (n^{\log_b a} \lg n)$.

3. If f(n) = $(n^{\log_b a + e})$ for some constant > 0, and if f(n/b) <= cf(n) for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta\ (f(n))$.

3.0 System Description

The system has the capability of producing unlimited practice problems. It does not rely on a prepared set of problems or a database of hardcoded problems with solutions. The problems in this system are generated at runtime. The solution is computed and matched with the answers provided by the user. The system then generates instantaneous feedback and grading for the answers submitted.

The underlying computation engine for all modules relies on substitution of integer values into equations that formulate in the steps of generating the solution.

The system is presented with a Graphical User Interface with multiple panels. The outer container of the UI is the Main Control Panel which contains five sub panels i.e. –

- Substitution Method Panel,
- Substitution Method Test Panel
- Master Theorem Panel
- Master Theorem Sub Panel
- Order of Growth Panel

Depending on what the user wants to view, the inner sub panels are instantiated and embedded in the outer Main Control Panel. This Main Control Panel is designed in such a way that it can host and change subpanels at runtime.

There are icons with tooltips on the top of the Main Control Panel which, when pressed instantiate the corresponding panel and launches its UI. By default the Substitution Method Panel is shown on startup of the system.

All problem panels share a similar UI design outlook. The similar functional areas of the problem panels are-

- Problem Panel – This area displays the problem generated by the system.
- Answer Panel – This area displays the options available to answer the problem.
- Instructions Panel – This area provides instructions to the user regarding how to approach the solution of the problem and how to answer it using the Answer section.
- Buttons Panel – This panel presents the following buttons for the user –
    - New Problem – It automatically generates a new problem.
    - Solution – It generates a solution to the problem using the computation engine and switches to the solution panel and displays the step wise solution to the problem.
    - Reset – It resets the Answer Panel selection and entries.
    - Grade – It submits the answers and compares them with the solution generated by the computation engine and provides credit and feedback for the submission.
- Solution Panel - The solution panel shows the generated solution to the problem.

The two test panels are also similar in their design outlook.

The test panels has four similar functional areas i.e.

- Recurrence Panel – This area updates the recurrence relation being created by the user.
- Setup Panel – This area provides an interface to the user for creating the recurrence and its testing bound.
- Solution Panel – This area displays the solution to the problem created above using the computation engine.
- Buttons Panel – This panel presents the following buttons for the user –
    - Add to f(n) – It adds the operations chosen in the Setup Panel to f(n) part of the recurrence.
    - Add to Guess - It adds the operations chosen in the Setup Panel to f(n) part of the bound in the Guess Panel.
    - Reset Recurrence – Clears whatever recurrence has been created by the user so far.
    - Reset Guess - Clears whatever bound has been created by the user so far.
    - Solve – It generates a solution to the problem using the computation engine and switches to the solution panel and displays the step wise solution to the problem.

3.1 Substitution Method Panel

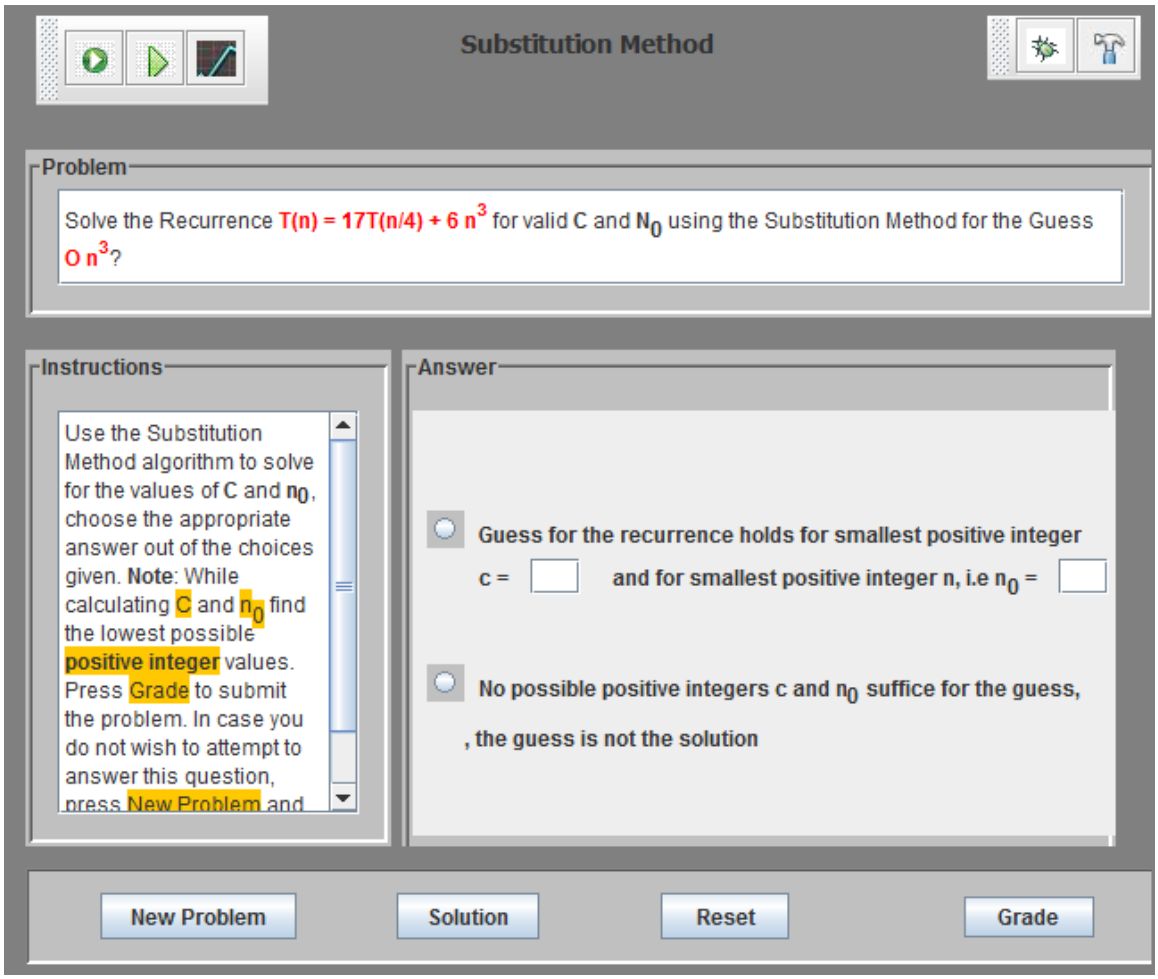

Figure 1. Substitution Method Panel

The Substitution Method Panel gives the user a practice area wherein recurrence relationships and their solutions generated by Substitution Method can be viewed. There are two approaches to learn from this module. Either the student can generate numerous recurrences by using the 'New Problem' button and then view the solution to the generated problem by clicking the 'Solution' button. This way the student can see different kinds of recurrence equations and their solutions. This increases the

understanding of how the substitution method works and also an understanding of recurrences and asymptotic bounds as such.

Alternatively, the student can test his grasp of the topic by submitting his answers to the generated problem for evaluation. The evaluation system generates feedback with the correct solution for the problem. The system also computes and displays grading for the submitted answer. This helps the students evaluate if they have understood the concepts well enough.

3.1.1 Problem

The Problem generated in the Substitution Method Panel is of the form –

Solve the Recurrence $T(n) = aT(n/b) + f(n)$ for valid c and n_0 using the Substitution Method for the Guess – g(n). In the recurrence, f(n) can be a polynomial function of n with lower order terms.

Using the Substitution Method algorithm the problem generated is solved for the values of c and n_0. The appropriate answer is chosen with values of c and n_0 filled out.

For O bound problems first the lowest positive integer c is found, then for that c the corresponding lowest positive integer n0 is found.

In the case of Ω bound problems first the largest positive integer c is found, then for that c the corresponding lowest positive integer n0 is found.

As the computation engine underlying this module is based on numerical substitution, it was simpler to keep numerical substitution into equations as integers to

provide better performance of the system and reduce considerable lag in solving these problems. It also made the User Interface simpler.

3.1.2 Problem Generation

The system generates a set of problems and then filters them to choose the problem statement which fits certain criteria, to be a feasible problem statement with not more than moderate difficulty.

There is high probability assigned to the generation of the first case given in the answering section of the panel. The generator first chooses for which option to generate a problem, then creates a problem set and filters the problems until a feasible problem is generated.

The problem uses the computation engine of the Master Theorem module to compute an asymptotic bound for the generated recurrence. This bound is then modified slightly to create the guess complexity of the problem. This ensures that the recurrence and the guess complexity in the problem statement are reasonable enough to be presented as a problem.

The constraints are modifiable through several constants which keep the generation of these problems very configurable. Some of the configuration options are:

- Keep value of c within the range of 5 and 20 if n_0 is 1
- Keep value of n_0 within the range of 5 and 20 if c is 1
- Probability of generating O or Ω as guess time complexity

- Number of terms to be generated in the guess time complexity function and f(n) part of the recurrence
- Type and probability of terms to generate in the functions both in guess time complexity function and f(n) part of the recurrence

3.1.3 Solution

Clicking on the Solution button invokes the computation engine to generate the solution for the problem statement. The problem is solves for the lowest integer values of c and n_0. The step by step solution is generated and displayed on the solution panel.

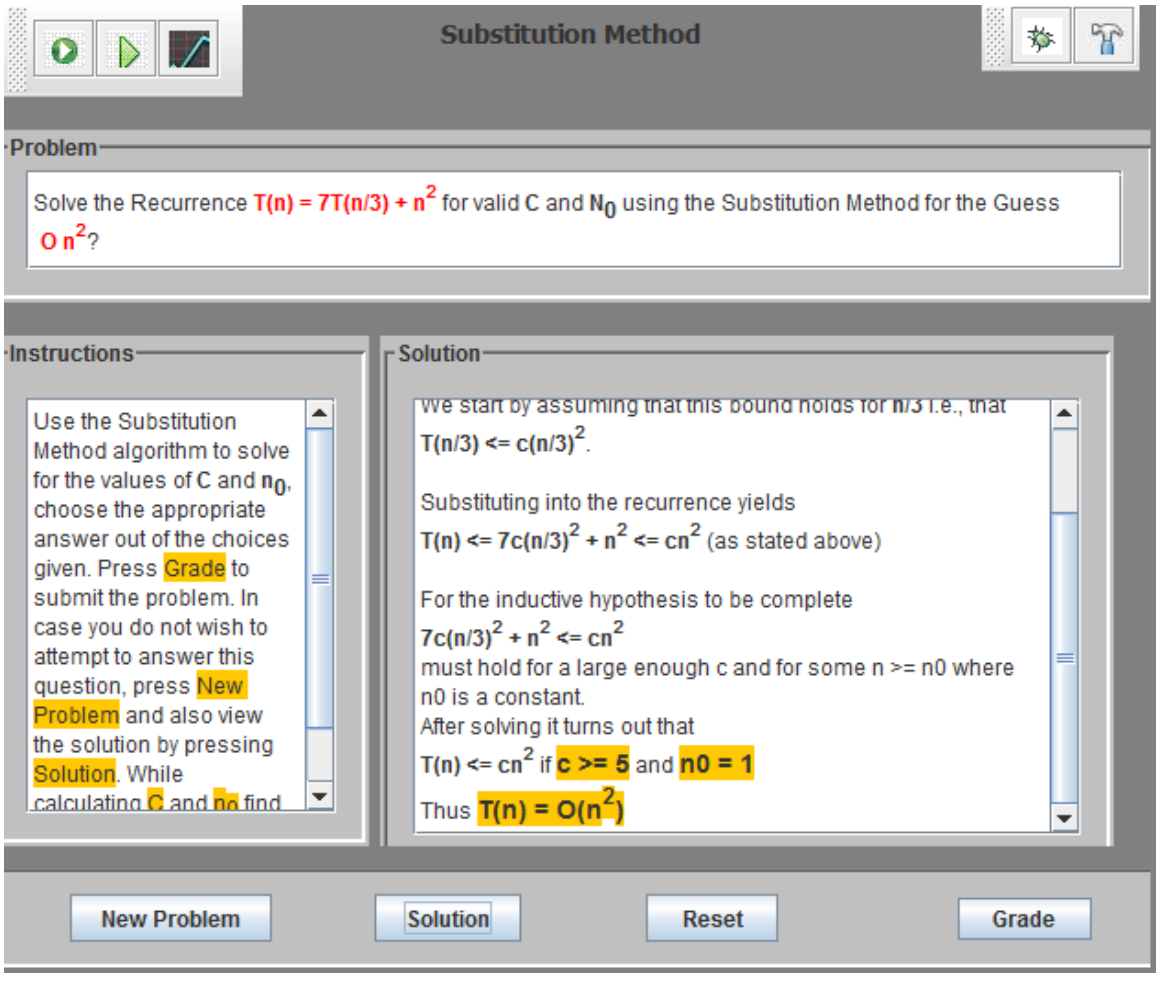

Figure 2. Substitution Method Solution Panel

3.1.4 Grading and Feedback

The system takes the answers given in the answering section and evaluates it with the answers generated by the computation engine.

Grading is based on the following criteria - .

Total grade for a correct answer is 100 (50 for c , 40 for n_0 and 10 for making the correct choice).

Grade for c and n_0 will be calculated respectively by the following formula: 100% if value matches computed value, 25% if it is 1 greater than correct c, 10% if it is 2 greater than correct c and 0% for all other c values.

So a correct submitted value will have full credit. Small deviation of +1 and +2 will get partial credit. The credits of n_0 will be dependent on the submitted value of c and not the value computed by the system. The system will compute the appropriate n_0 using the submitted c. In case the submitted n_0 does not match this n_0 computed by the system, 0% credit will be given. 100% credit for n_0 will be granted if the value of n_0 computed by the system matches the n_0 submitted.

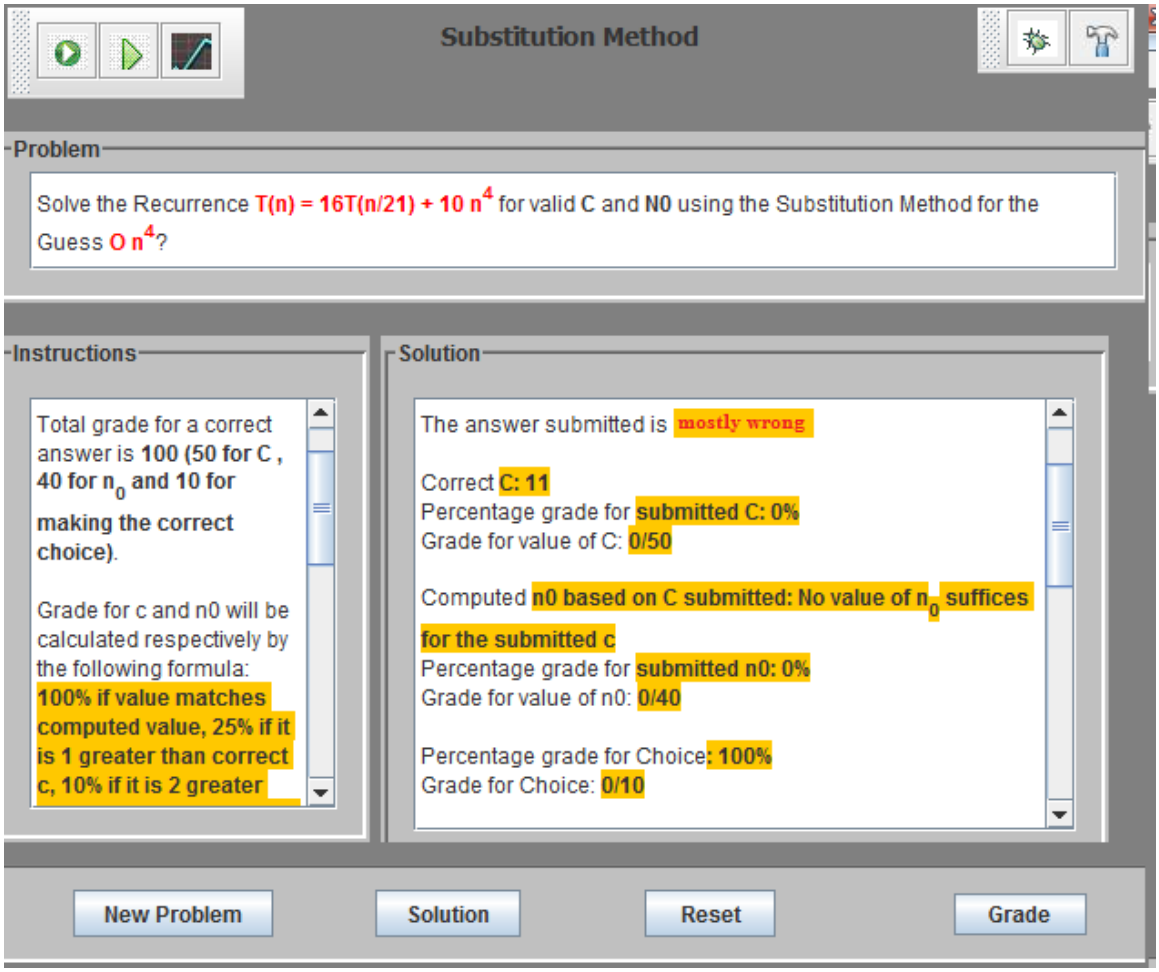

Figure 3. Substitution Method Grade Panel

A sample problem generated by the system with its solution and evaluation is shown below:

Problem Generated:

Solve the Recurrence $T(n) = 16T(n/21) + 10\ n^4$ for valid c and n_0 using the Substitution Method for the Guess $O\ n^4$?

Evaluation:

Your Answer: Guess for the recurrence holds for the smallest positive integer c = 1 and for smallest positive integer n, i.e $n_0 = 1$

The answer submitted is largely wrong

Correct c: 11

Percentage grade for submitted c: 0%

Grade for value of C: 0/50

Computed n_0 based on c submitted: No value of n_0 suffices for the submitted c

Percentage grade for submitted n_0: 0%

Grade for value of n_0: 0/40

Percentage grade for Choice: 100%

Grade for Choice: 0/10

Total Grade for Problem: 10/100

Please see the solution below.

**********Solution**********

Guessing the solution is $T(n) = O(n^4)$ which means that $T(n) <= cn^4$ for an appropriate choice of the constant $c > 0$.

We start by assuming that this bound holds for n/21 i.e., that $T(n/21) <= c(n/21)^4$.

Substituting into the recurrence yields

$T(n) <= 16c(n/21)^4 + 10\ n^4 <= cn^4$ (as stated above)

For the inductive hypothesis to be complete

$16c(n/21)^4 + 10\ n^4 <= cn4$

must hold for a large enough c and for some $n >= n_0$ where n_0 is a constant.

After solving it turns out that

$T(n) <= cn^4$ if $c >= 11$ and $n_0 = 1$

Thus $T(n) = O(n^4)$

3.2 Substitution Method Test Panel

The system also provides a test panel wherein the user can create custom recurrence equations and test them against custom created bounds.

This panel is useful to test the same recurrence against changing upper or lower bounds to understand how c and n_0 vary with changing bounds.

The UI for this panel has the four functional areas mentioned earlier plus a fifth functional are i.e.

- Guess Panel – This area updates the bound being created by the user.

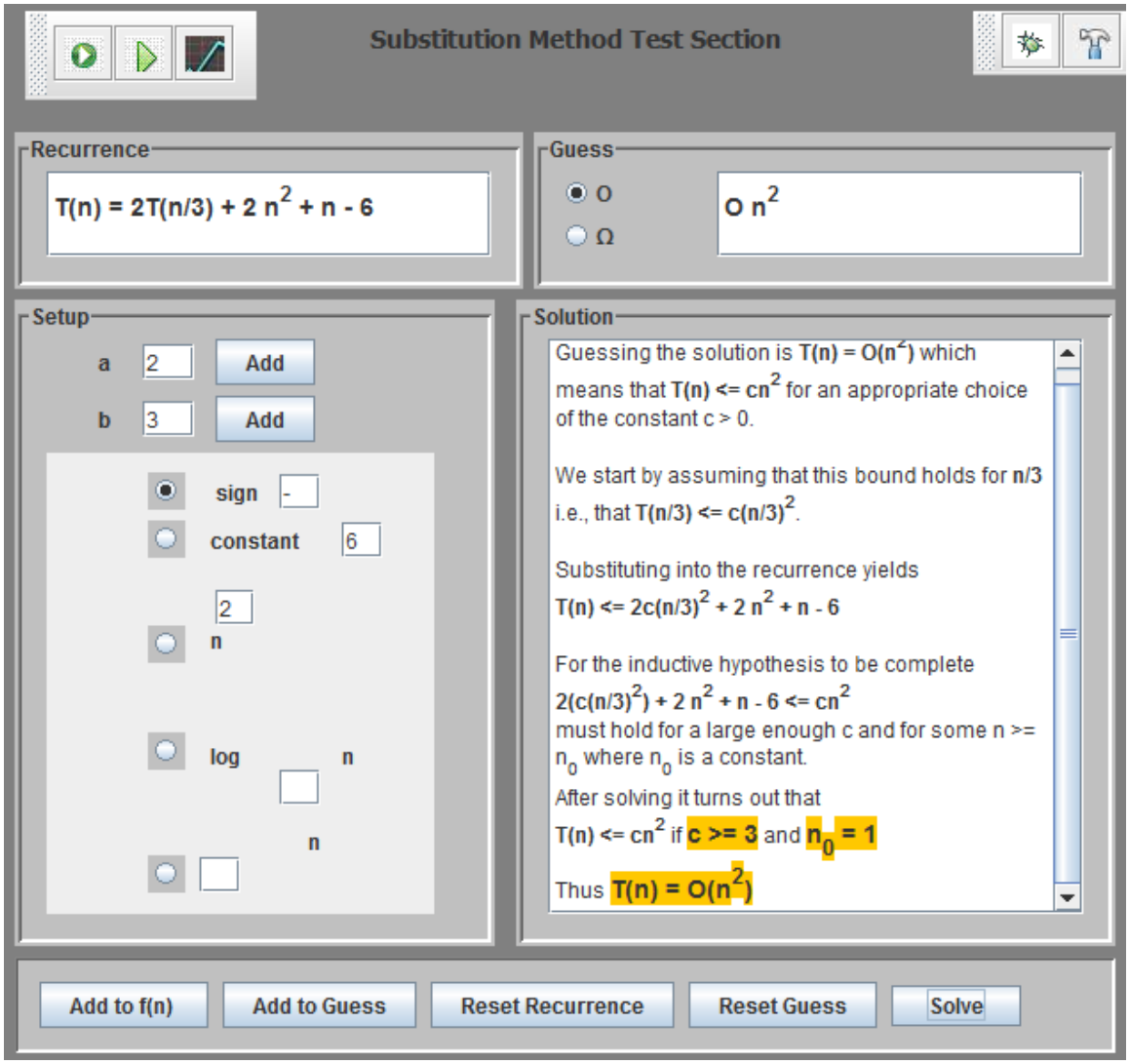

Figure 4. Substitution Method Test Panel

3.3 Master Theorem Panel

The Master Theorem Panel gives a practice area to solve simulated exercises generated to impart the learning of the Master Theorem. The module generates Recurrence relationship problems and similar to the Substitution Method Panel there are two approaches to learning from them.

The student can either generate numerous problems and view their auto generated solution or choose to test their understanding of the topic by submitting their answers and letting the system evaluate them.

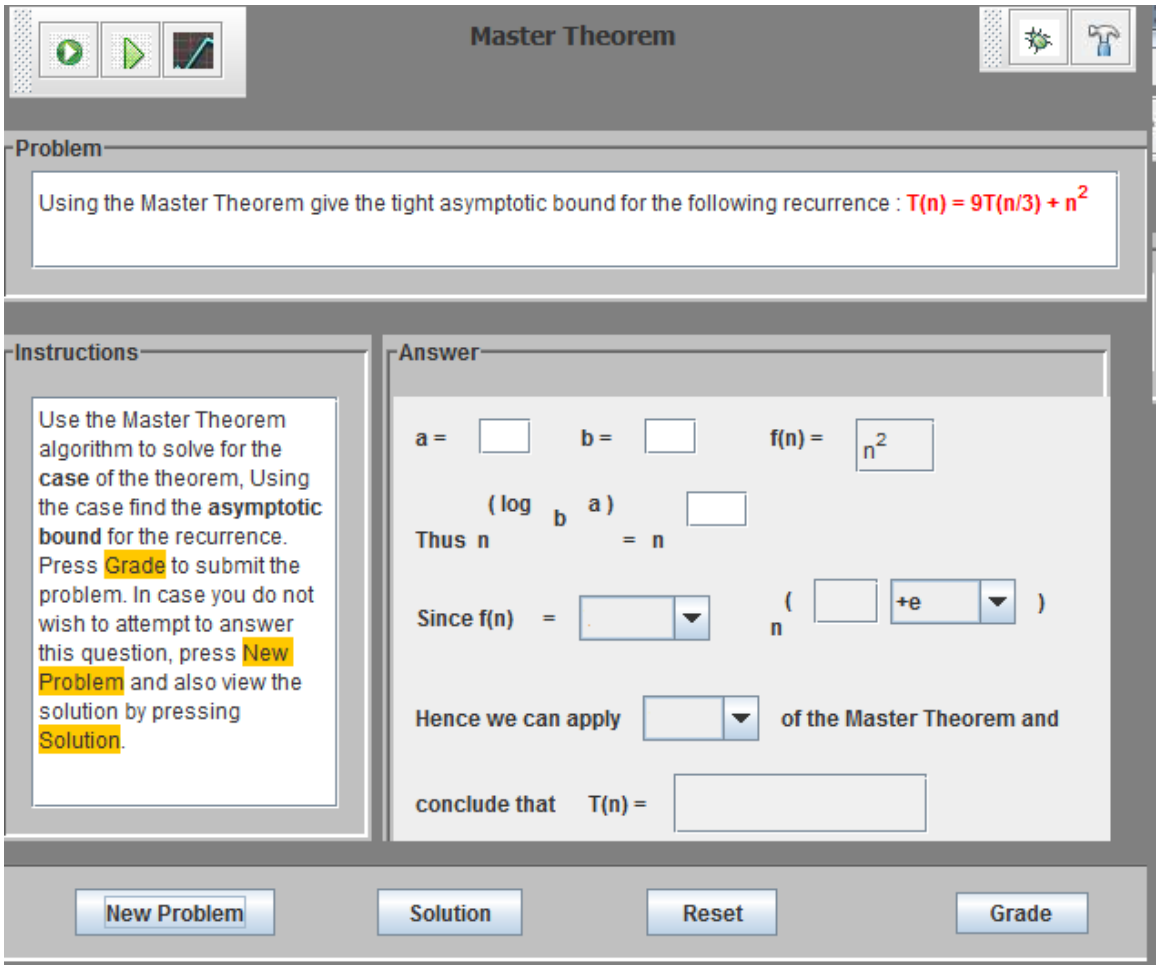

Figure 5. Master Theorem Main Panel

3.3.1 Problem

The problem displayed on the Master Theorem Panel is of the form –

Using the Master Theorem give the tight asymptotic bound for the following recurrence :

$T(n) = aT(n/b) + f(n)$

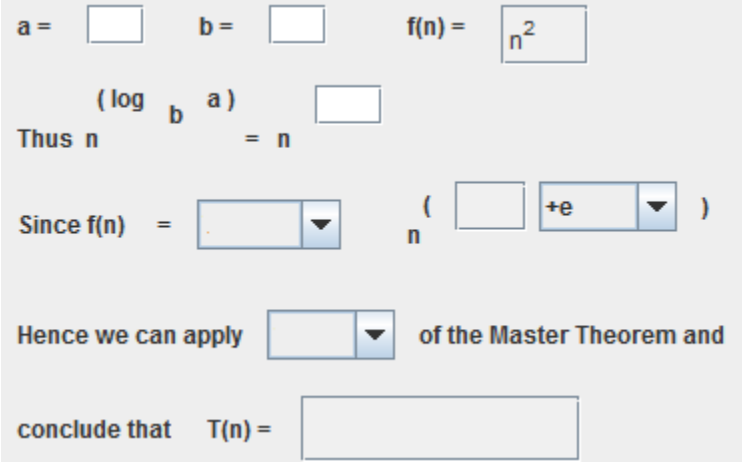

Figure 6. Master Theorem Answer Panel

In order to answer the problem the Answer panel provides certain input and combo boxes for the user to input. The user is made to select a series of choices as well as input certain numeric values as his answer. His choices and input values are then converted internally to the asymptotic bound that the user is suggesting through their choices.

The values to be input are

- a
- b
- $n^{(\log_b a)}$ – Here the value is matched up to only two sequential digits after the decimal.

After entering these values, the combo boxes present options to choose from. According to the choices made the complexity to be submitted it computed dynamically and is updated in the 'T(n) =' box.

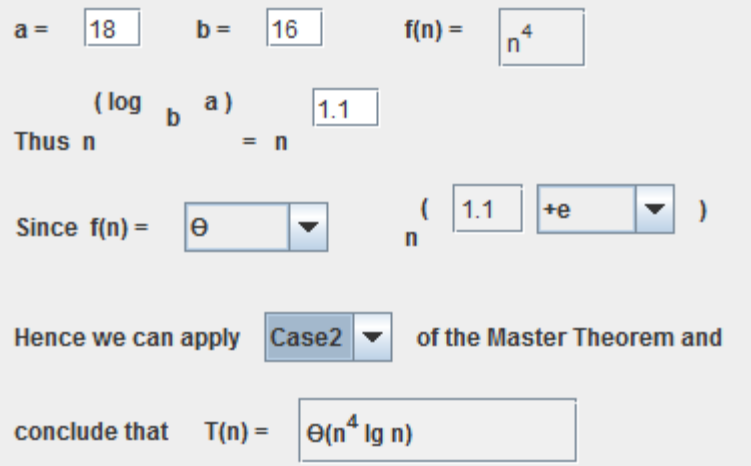

Figure 7. Master Theorem Answer Section Completed

3.3.2 Problem Generation

The system works similar to the Substitution Method problem generator. It generates a set of problems and then filters them to choose a problem statement which fit desired criteria and appears to be a feasible problem statement with not more than moderate difficulty.

There is almost equal probability assigned to the generation of the three cases given in the Master Theorem. The generator firstly chooses for which option to generate a problem, then creates a problem set and filters it until a feasible problem is generated.

The configuration values are similar as used by Substitution Method and are reused from that module.

3.3.3 Solution

Clicking on the Solution button invokes the Master Theorem computation engine to generate the solution for the problem statement. Once the case of the theorem is

computed, the correct asymptotic bound is calculated by the engine. The step by step solution is generated and displayed on the solution panel.

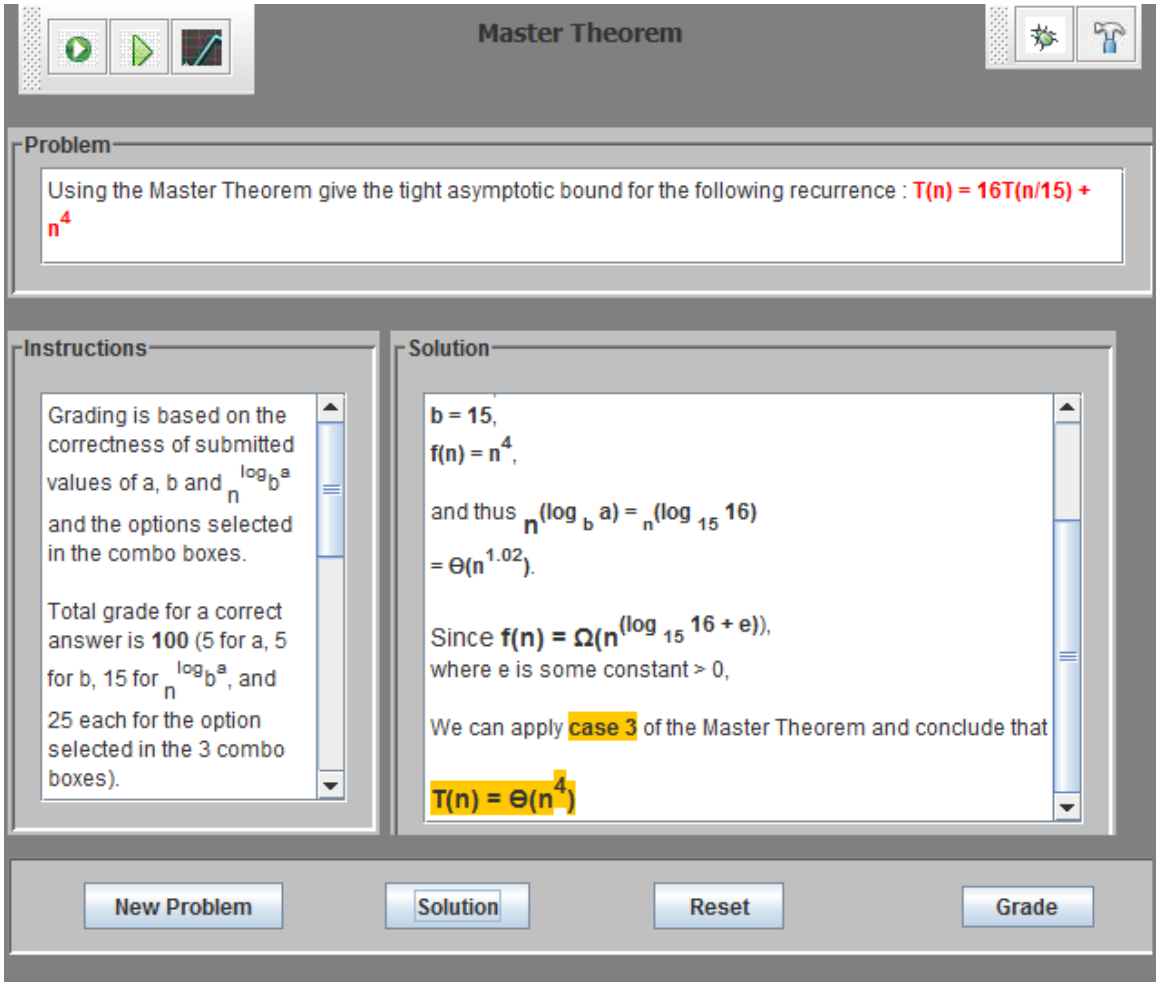

Figure 8. Master Theorem Solution Panel

3.3.4 Grading and Feedback

The system takes the answers given in the answering section and evaluates it against the answers generated by the computation engine.

Grading is based on the correctness of submitted values of a, b and $n^{\log_b a}$ and the options selected in the combo boxes.

Total grade for a correct answer is 100 (5 for a, 5 for b, 15 for $n^{\log_b a}$, and 25 each for the option selected in the 3 combo boxes).

Grade for the a, b and the options selected in the combo boxes will get 0 credit on any deviation from correct answer.

So a correct submitted value will have 100% and hence full credit.

Grade for $n^{\log_b a}$ will be calculated by the following formula:

100 - |%deviation from the value|) * (percentage weightage of that element)

i.e. 100 - (|correct val - submitted val|)/correct val *100) * (percentage weightage of that element)

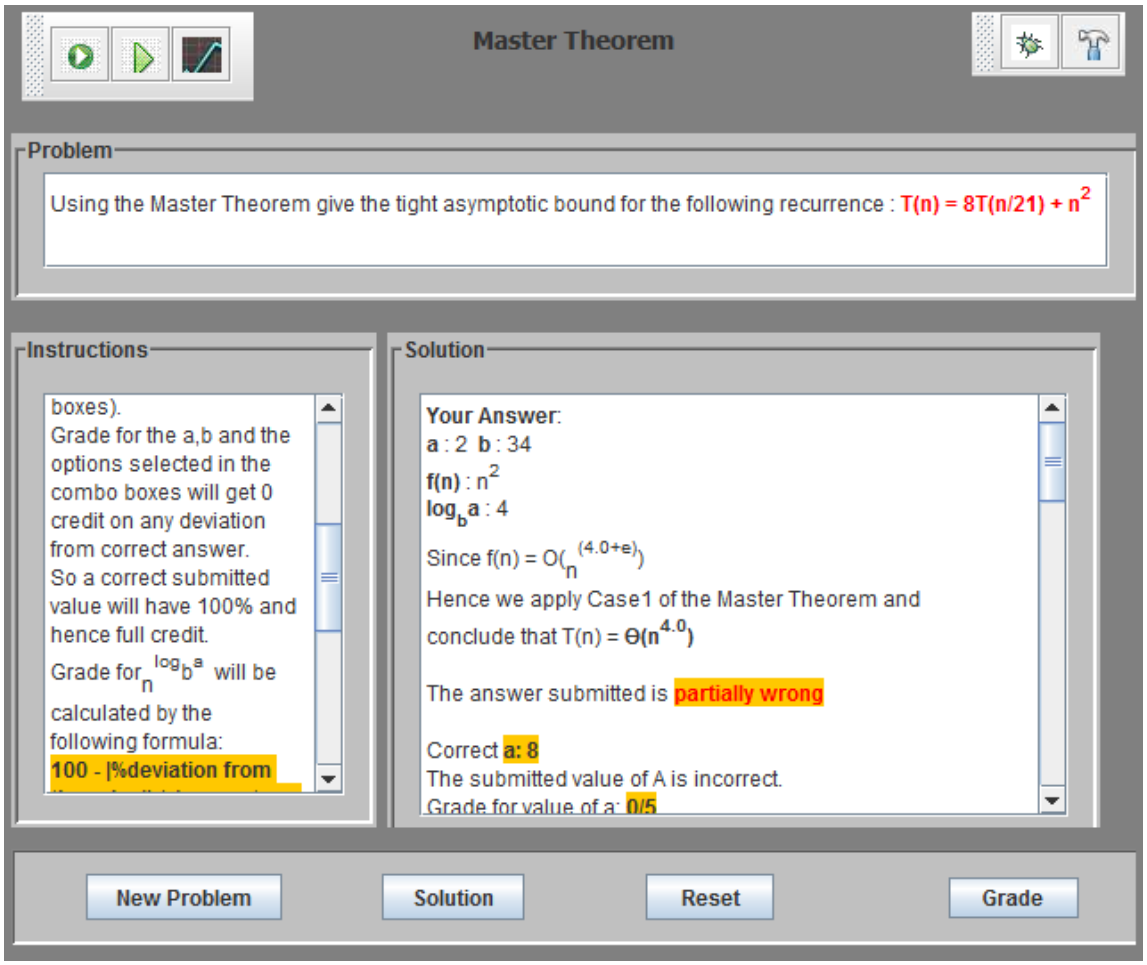

Figure 9. Master Theorem Grade Panel

A sample problem generated by the system with its solution and evaluation is shown below:

Problem Generated:

Using the Master Theorem give the tight asymptotic bound for the following recurrence : $T(n) = 16T(n/2) + n^4$

Evaluation:

Your Answer:

a : 16 b : 2

f(n) : n^4

$\log_b a$: 4

Since $f(n) = \Theta(n^{(4.0+e)})$

Hence we apply Case2 of the Master Theorem and

conclude that $T(n) = \Theta(n^4 \lg n)$

The answer submitted is almost correct.

Correct a: 16

The submitted value of A is correct.

Grade for value of a: 5/5

Correct b: 2

The submitted value of B is correct.

Grade for value of b: 5/5

Correct $\log_b a$: 4

Deviation of submitted $\log_b a$: 0%

Grade for value of $\log_b a$: 15/15

Correct bound type of f(n): Θ

The submitted bound type of f(n) is correct.

Grade for value of bound type of f(n): 25/25

Correct constant term to be added/subtracted in the exponent :

The submitted constant term to be added/subtracted in the exponent is incorrect.

Grade for constant term to be added/subtracted in the exponent: 0/25

Correct case of Master Theorem: Case 2

The submitted case of Master Theorem is correct.

Grade for case of Master Theorem: 25/25

Total Grade for Problem: 75/100

Please see the solution below.

**********Solution**********

Solution by Master Theorem -

For this recurrence, we have

$a = 16,$

$b = 2,$

$f(n) = n^4,$

and thus

$n(\log b\ a) = n(\log_2 16)$

$= \Theta(n^4).$

Since $f(n) = \Theta(n^{(\log_2 16)})$,

we can apply case 2 of the Master Theorem and conclude that

$T(n) = \Theta(n^{\log_2 16} \lg n)$ or

$T(n) = \Theta(n^4 \lg n)$

3.4 Substitution Method Test Panel

The system also provides a test panel wherein the user can create custom recurrence equations and compute their asymptotic complexities using the Master Theorem computation engine.

This module enables learning through creating fully configurable recurrences and observing their tight bounds being auto generated by the computation engine.

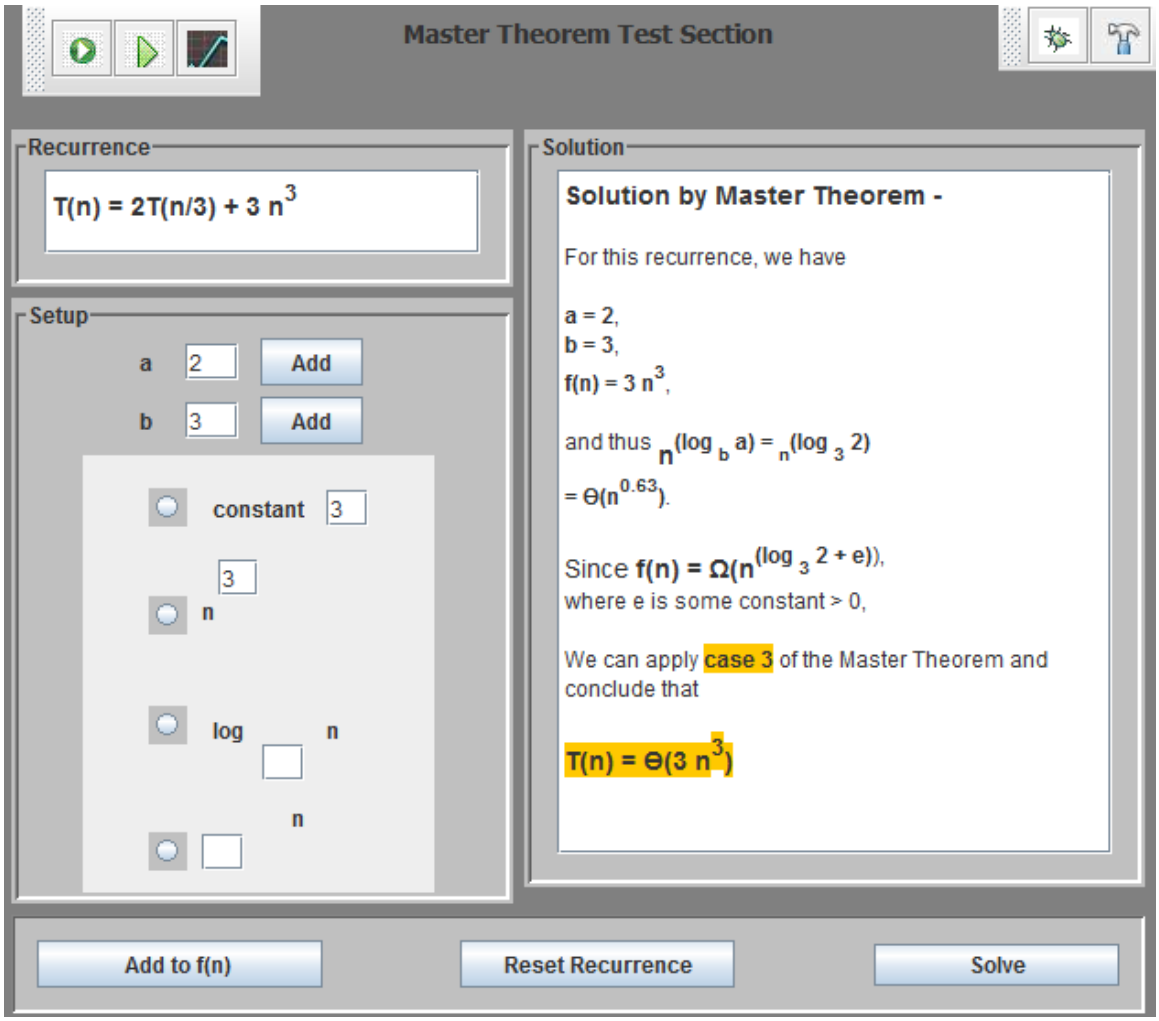

Figure 10. Master Theorem Test Panel

3.5 Order of Growth Panel

The Order of Growth panel provides simulated exercises which are helpful in gaining a clear idea of how various asymptotic bounds compare to each other. The system has the capability of generating unlimited function sets with a new set of functions appearing with in every new problem. Similar to the previous problem panels, the student can either choose to see the solutions to the problems being generated or test

their understanding of order of growths by ranking the functions and submitting the ranked sequence for evaluation.

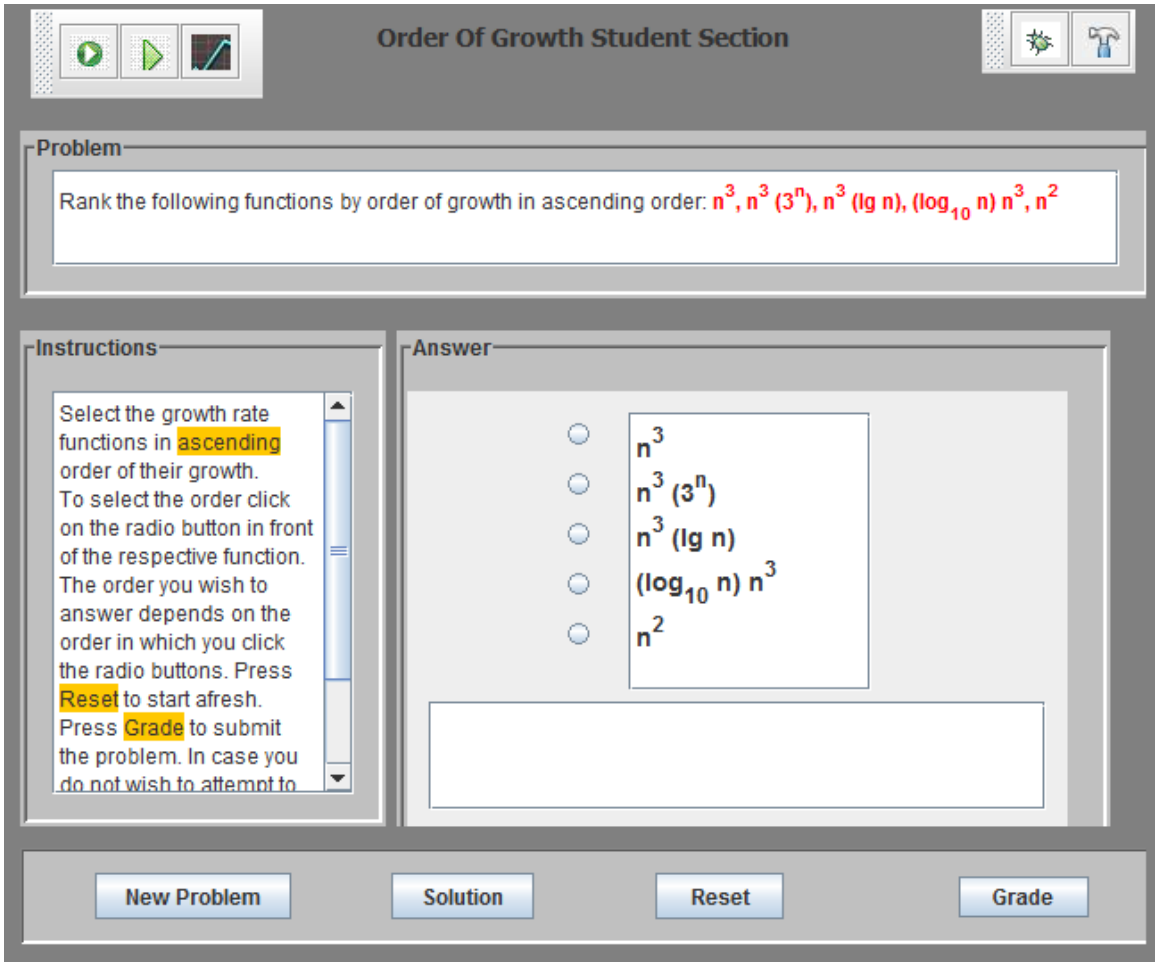

Figure 11. Order of Growth Panel

3.5.1 Problem

The Problem generated in the Order of Growth Panel is of the form –

Rank the following functions by order of growth in ascending order: f1(n), f2(n), f3(n), f4(n), f5(n).

Using the Order of Growth Comparator module arrangement of the functions is done in ascending order of growth such that f1(n) = Ω (f2(n)), f2(n) = Ω ((f3(n)), ... f4(n) = Ω (f5(n)).

In order to answer the problem the Answer panel provides the functions with radio button in front of them to select. The order of functions intended to be submitted is order in which the radio buttons are selected. The selected order is above the submit button. The ranking process can be restarted by clicking the reset button.

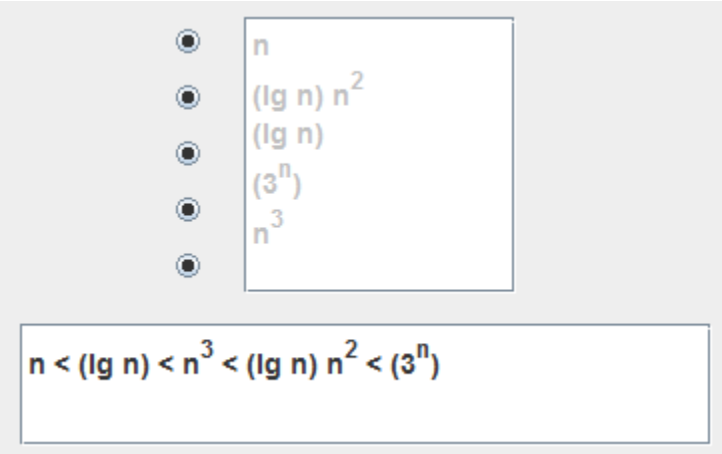

Figure 12. Order of Growth Answer Panel

3.5.2 Problem Generation

The system generates functions according to criteria specified in a configuration file. The system makes sure that no two functions of the same order of growth are generated.

The constraints are modifiable through constants which keep the generation of these problems configurable. Some of the configuration options are like

- No of terms to be generated in the guess time complexity function and f(n) part of the recurrence
- Type and probability of terms to generate in the functions

3.5.3 Solution

Clicking on the Solution button invokes the computation engine to generate the solution for the problem statement. The functions are sorted in ascending order of growth and displayed on the solution panel.

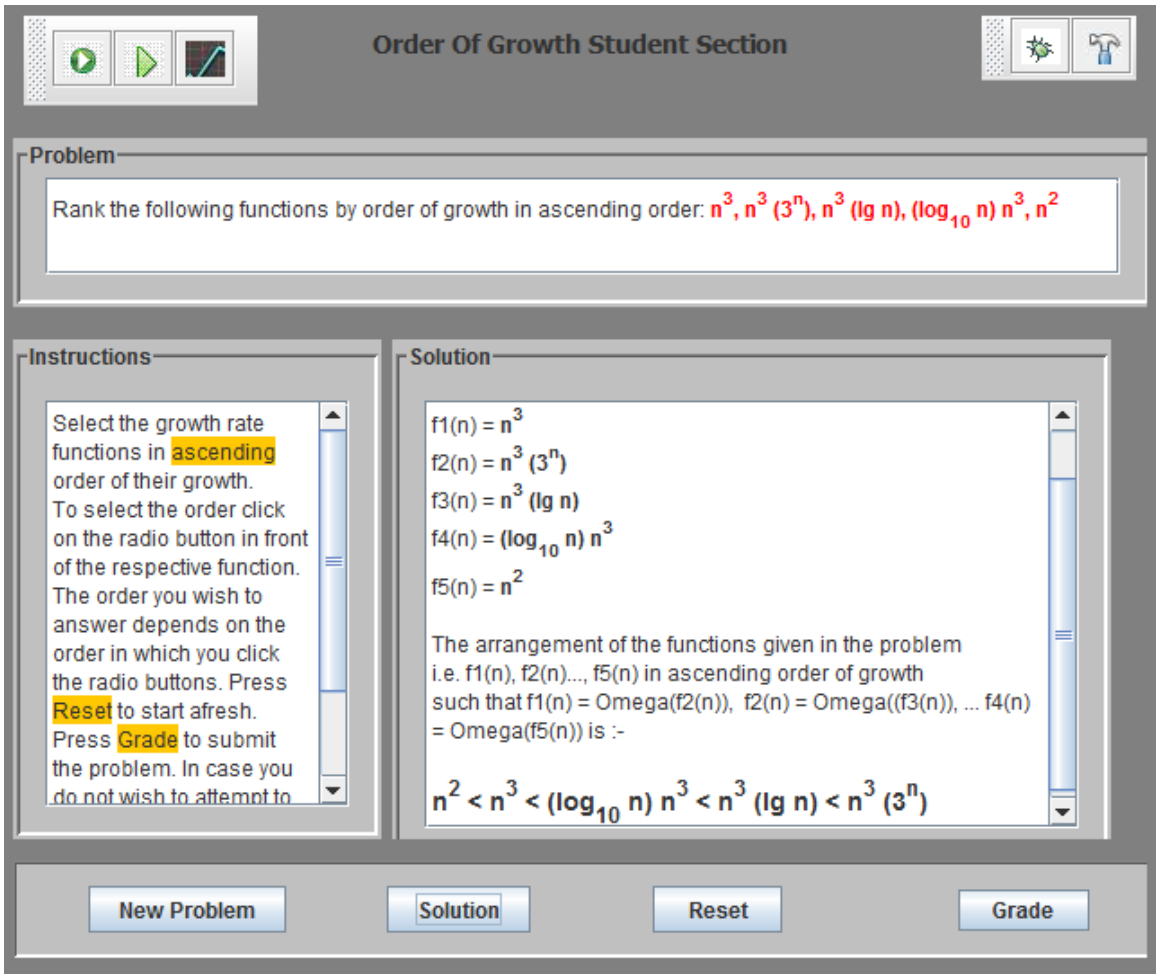

Figure 13. Order of Growth Solution Panel

3.5.4 Grading and Feedback

The system takes the answers given in the answering section and evaluates it with the answers generated by the computation engine.

There are 5 functions. When you put all 5 in order, it implies an order on ${}^5C_2 = 10$ pairs of functions i.e. if you have a < b < c < d < e, it implies the following pairs: a<b, a<c, a<d, a<e, b<c, b<d, b<e, c<d, c<e, d<e.

Total grade for a correct answer is 100. Each matching inequality pair will get 10 credit amounting to a total of 100 for all 10 correct inequalities.

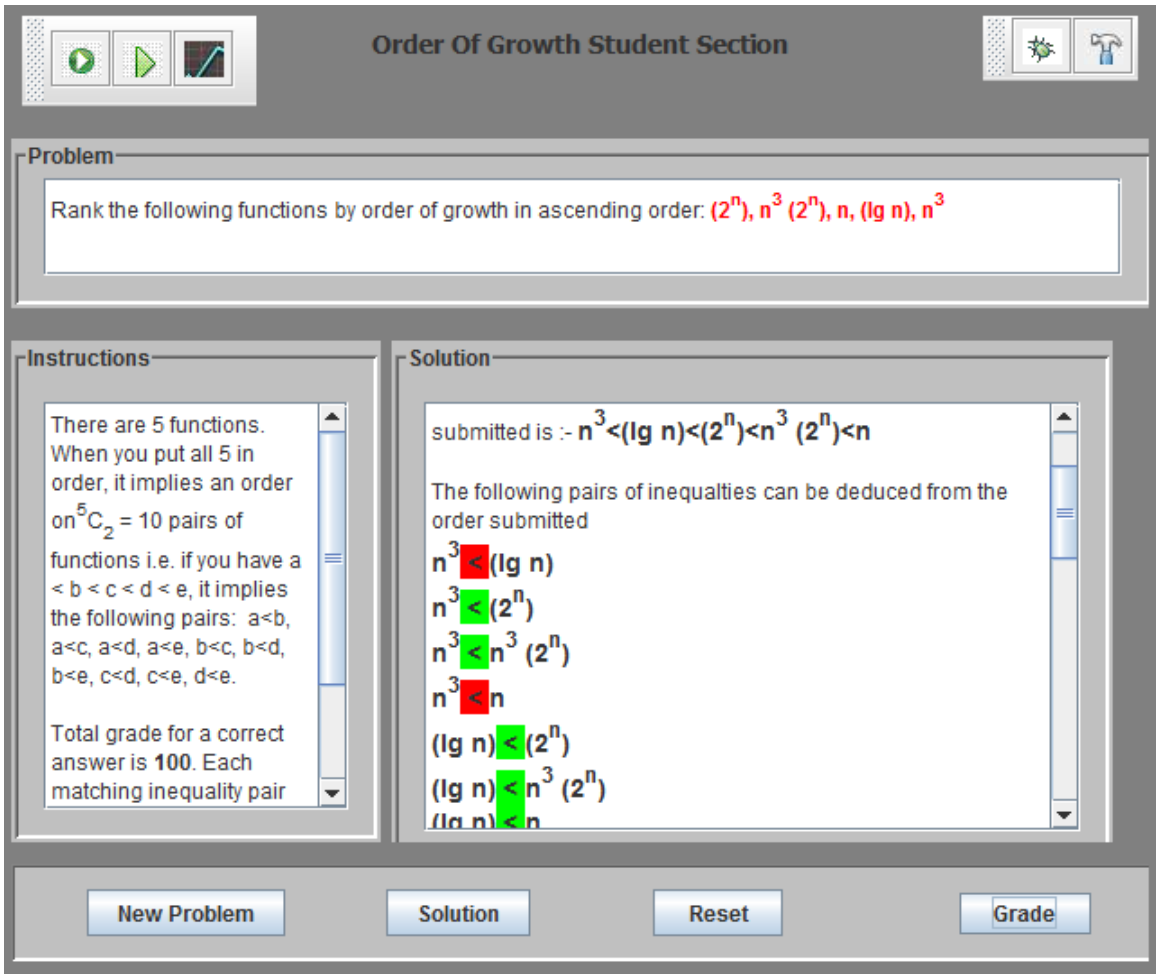

Figure 14. Order of Growth Grade Panel

A sample problem generated by the system with its solution and evaluation is shown below:

Problem Generated:

Rank the following functions by order of growth in ascending order: (2^n), n^3 (2^n), n, (lg n), n^3

Evaluation:

Your Answer: The arrangement of the functions given in the problem

i.e. f1(n), f2(n)..., f5(n) in ascending order of growth

such that $f1(n) = \Omega(f2(n))$, $f2(n) = \Omega((f3(n))$, ... $f4(n) = \Omega(f5(n))$

submitted is :- $n^3 < (\lg n) < (2^n) < n^3\ (2^n) < n$

The following pairs of inequalities can be deduced from the order submitted

$n^3 < (\lg n)$

$n^3 < (2^n)$

$n^3 < n^3\ (2^n)$

$n^3 < n$

$(\lg n) < (2^n)$

$(\lg n) < n^3\ (2n)$

$(\lg n) < n$

$(2^n) < n^3\ (2^n)$

$(2^n) < n$

$n^3\ (2^n) < n$

The inequalities above in red are wrong and the ones in green are correct.

The answer submitted is partially wrong.

Of the arrangement submitted 6 pairs of inequalities are correct.

Total Grade for Problem: 60/100

Please see the solution below for correct order.

**********Solution**********

Given functions:

$f1(n) = (2^n)$

$f2(n) = n^3 (2^n)$

$f3(n) = n$

$f4(n) = (\lg n)$

$f5(n) = n^3$

The arrangement of the functions given in the problem

i.e. f1(n), f2(n)..., f5(n) in ascending order of growth

such that $f1(n) = \Omega(f2(n))$, $f2(n) = \Omega((f3(n))$, ... $f4(n) = \Omega(f5(n))$ is :-

$(\lg n) < n < n3 < (2n) < n3 (2n)$.

4.0 Conclusion

In this project there was implementation of an Automatic Assessment Algorithm Analysis system which caters to the teaching of Recurrence Relationships and a better understanding of asymptotic time complexities which helps in imparting strong mathematical foundation to computer science students.

The system is a novel way for presenting automatically generated simulation exercises with auto solution generation and assessment of recurrence based problems.

The ability to generate unlimited worked examples with instant feedback fosters a rich learning environment for the students.

The implementation of the computation engines for solving recurrences through Substitution Method and Master Method can be utilized for more than just simulation exercises. The design of the object model used for the implementation can be reused for creating further enhanced recurrence solving tools and research on the analysis of recurrences.

APPENDIX A – Architectural Design

This section describes the detailed design of how the Algorithm Analysis System for Recurrence Relationships will be implemented. The section contains the class models used for various algorithm simulations.

Technologies/Architecture Used

The project will be developed using the Java Platform. For creating the GUI's Java Swing will be used.

The project will be designed using the Model View Controller Architecture. The core algorithm implementation classes will constitute the Model. The Controller will control the action events captured from the User Interface and passed on to the underlying algorithm implementation. The Controller would modify the view depending upon the choice of view selected by the user. The View generates be the GUI.

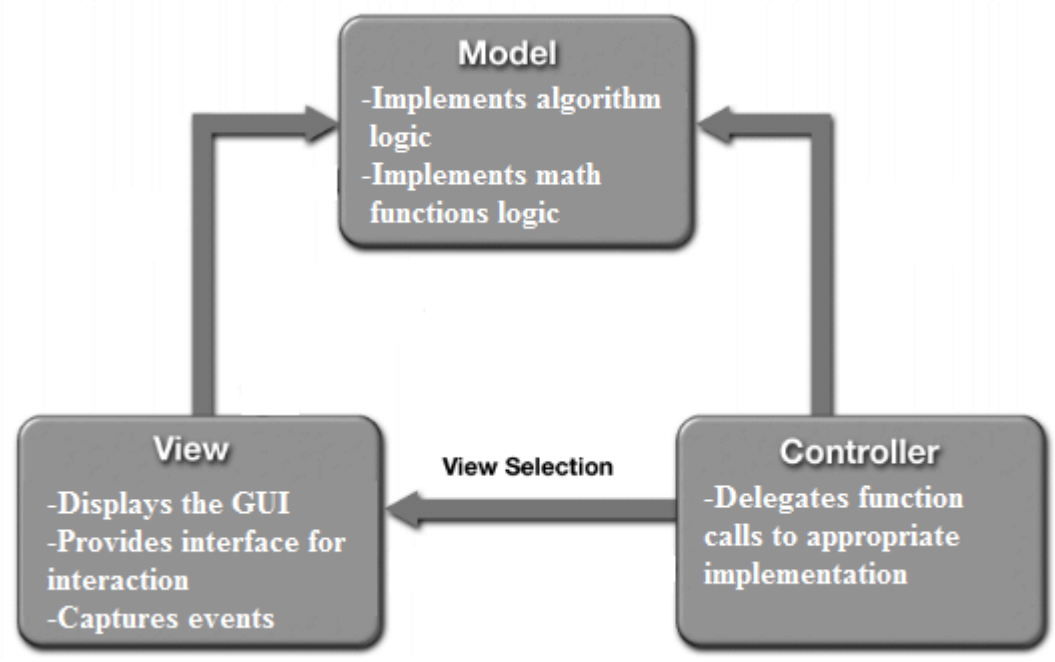

Figure 15. Architecture Diagram

Framework Design

The system is developed using Java Applets. Java Applets can utilize Java Swing library and thus can be used to build interactive GUIs and utilize all features of standalone Java development.

This section describes the design of the various Entity classes used in the system. For building a simulation engine of the algorithms for this system, there needs to be design of a math functions library. This library would provide the base for the execution of the simulated algorithms.

Math Operations Design

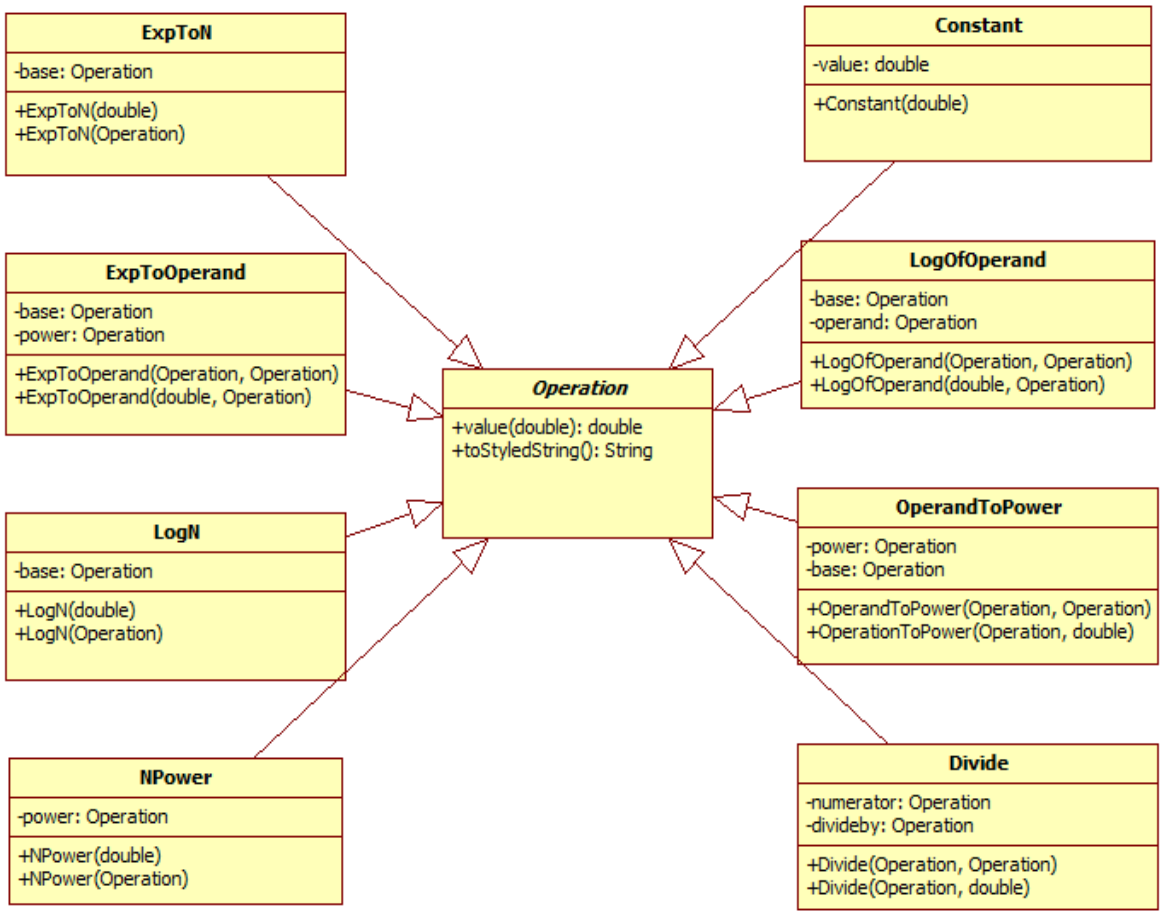

Figure 16. Math Operation Classes

In order to provide runtime polymorphism and use a generic Math Operation object reference to point to various Operation implementations, the abstract class Operation is defined. This abstract class defines two abstract functions i.e. value(double n) and toStyledString().

The value function takes as the argument the input operand for the function f(n) and returns the value of the function on the input n. All Operation implementation classes must override this abstract function and provide their own mathematical implementation of the function they are providing the functionality for.

The toStyledString() function is similar to Java's own toString() function. It provides a textual representation of the Operation Object with some styling tags embedded in it. These styling tags are processed by the Text Processor module written for the purpose of this system. The Text Processor module takes the output if the toStyledString() function and using the styling tags embedded in the text creates an object of java.swing.text.StyledDocument object.

All Math Operation implementing classes must inherit from this abstract class and provide implementations for both these functions.

Recurrence Equation Design

Representation of the Recurrence Equation requires design of some helper entity classes. The recurrence is basically of the form

$$\mathbf{T(n) = aT(n/b) + f(n)}$$

In order to represent this equation in code we require the classes shown in Figure17.

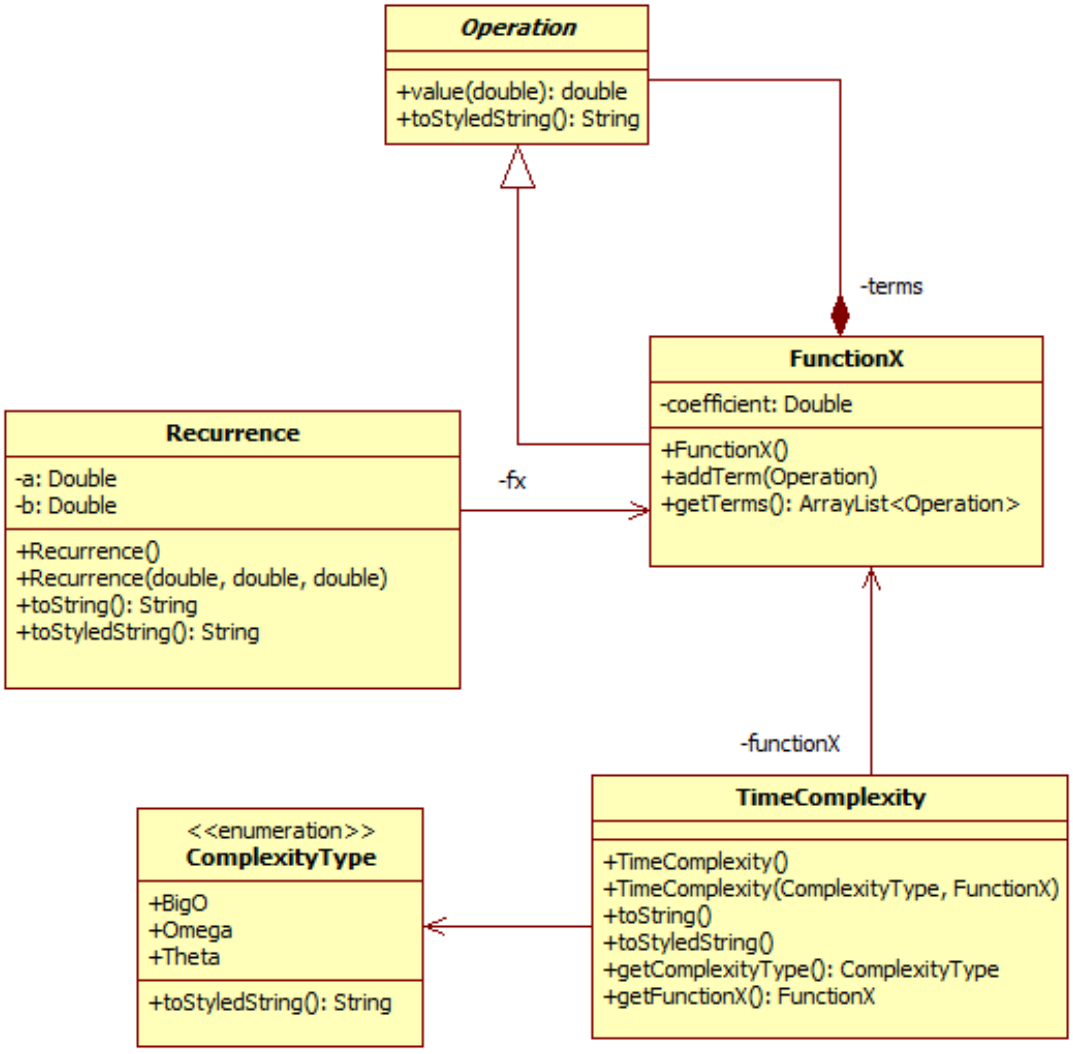

Figure 17. Recurrence Equation Classes

The Recurrence class contains the value of a, b and holds an object of type FunctionX. The class FunctionX is designed to contain inherits from Operation. It also contains an ArrayList of Operation objects. These objects can be of any class inheriting from Operation shown in Figure17. The recurrence class contains an object each of FunctionX and ComplexityType enum.

To represent the asymptotic bounds for the recurrence equations, the class TimeComplexity is used.

The recurrence class is an integral object used in the recurrence solving algorithm's implementation classes.

Substitution Method Computation Design

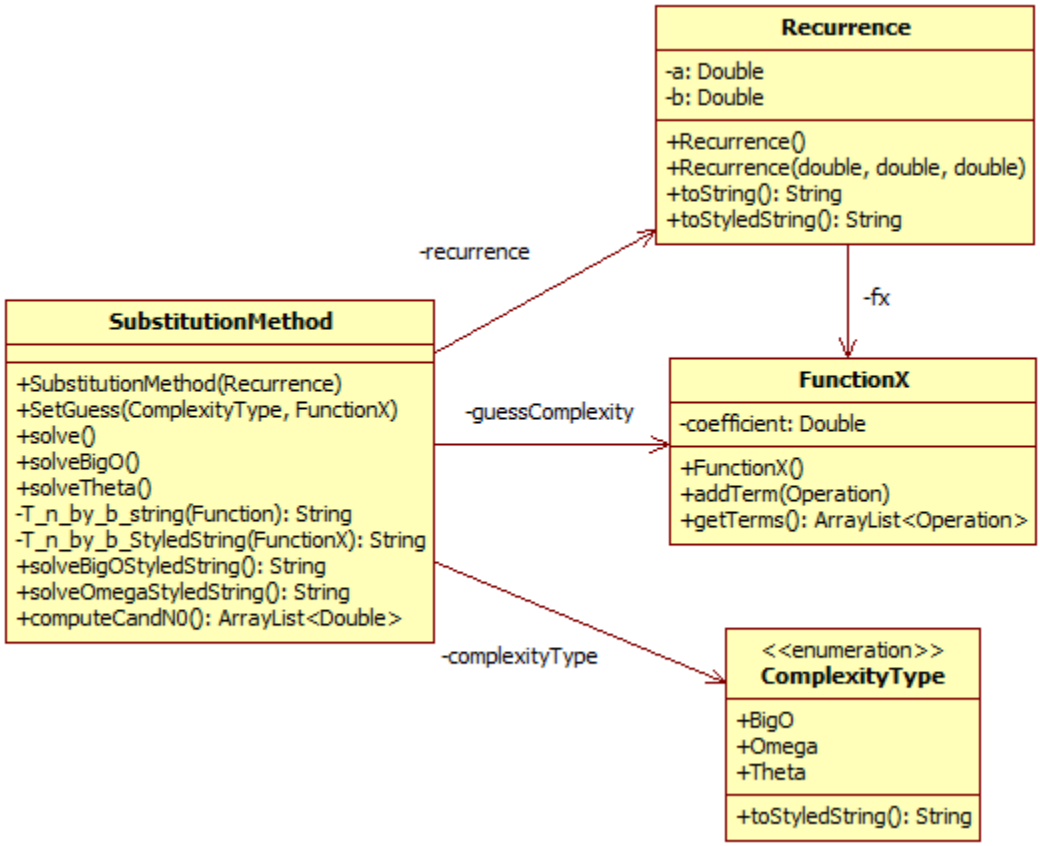

Figure 18. Substitution Method Simulation Classes

The substitution method algorithm is simulated using the SubstitutionMethod class. It takes in an object of Recurrence type and performs simulation on the bound guessComplexity and computes the n_0 and c for the problem presented.

Master Theorem Computation Design

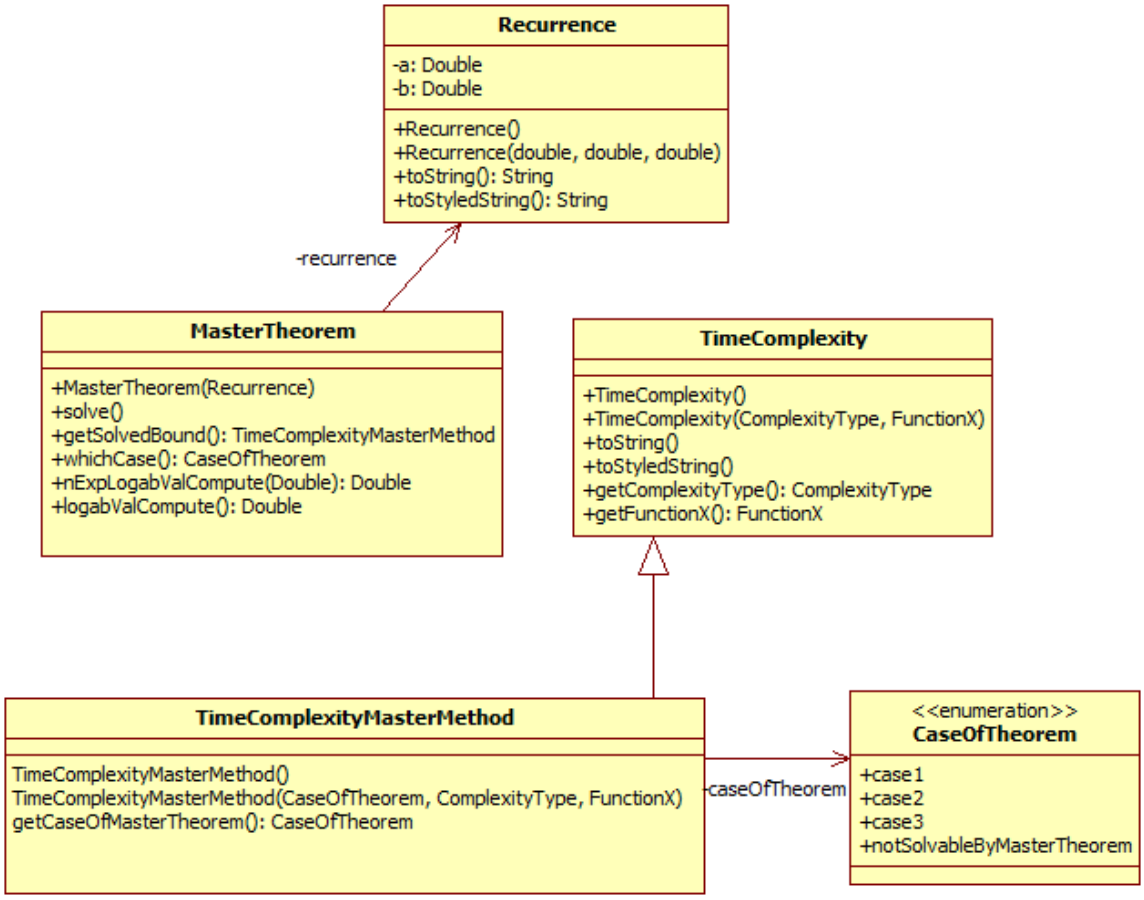

Figure 19. Master Theorem Simulation Classes

The Master Theorem is simulated using the MasterTheorem class. The constructor of this class takes an object of Recurrence type. The solve functions computes the case of the Master Theorem this recurrence lies false in. After it computes the case, it applies the theorem and computes the bound. The bound is returned as an object of TimeComplexityMasterMethod type which inherits from TimeComplexity class and contains an object of caseOfTheorem enum.

Order of Rate of Asymptotic Growth Functions

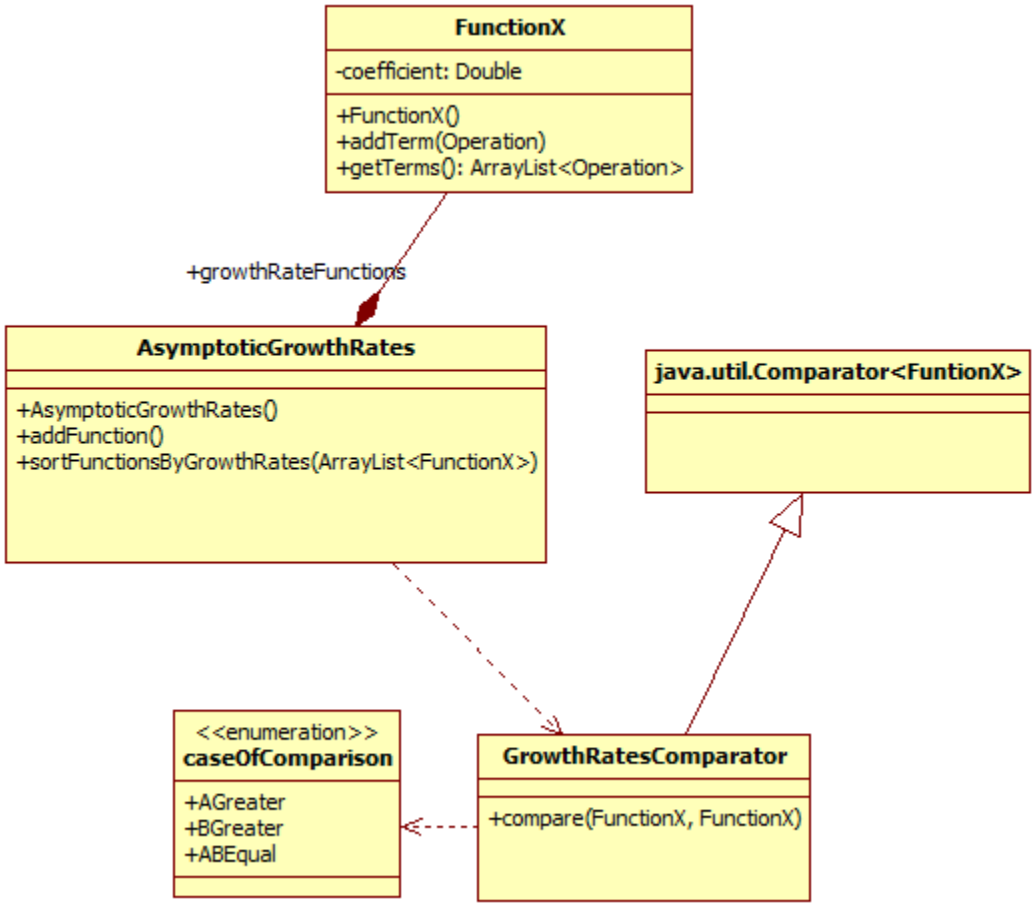

Figure 20. Order of Rate of Growth Classes

In order to be able to compare the Order of Rate of Asymptotic Growth Rates of Functions the class AsymptoticGrowthRates is designed. It contains a list of Functions represented as FunctionX objects. The comparison logic goes into the compare() function of the GrowthRatesComparator class which inherits from the generic java class Comparator.

APPENDIX B – Source Code

File: AsymptoticGrowthRates .java

```
package algorithmLibrary;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

public class AsymptoticGrowthRates {

	private ArrayList<FunctionX> originalFunctions;
	private ArrayList<FunctionX> growthRateFunctions;


	public ArrayList<FunctionX> getOriginalFunctions() {
		return originalFunctions;
	}

	public ArrayList<FunctionX> getGrowthRateFunctions() {
		return growthRateFunctions;
	}



	public AsymptoticGrowthRates()
	{
		originalFunctions = new ArrayList<FunctionX>();
		growthRateFunctions = new ArrayList<FunctionX>();
	}

	public AsymptoticGrowthRates(ArrayList<FunctionX> functions)
	{
		originalFunctions = new ArrayList<FunctionX>();
		growthRateFunctions = new ArrayList<FunctionX>();
		for (FunctionX fx : functions) {
			growthRateFunctions.add(fx);
			originalFunctions.add(fx);
		}
	}

	public void addFunction(FunctionX function)
	{
		growthRateFunctions.add(function);
	}

	public void printAllFunctions(ArrayList<FunctionX> functions)
	{
		int numFuntionsInARow = 5;

		for(int i = 1 ; i <= functions.size() ; i++)
		{
			if(i%numFuntionsInARow == 1)
			{
```

```
                                System.out.print("\n");

                        }
                        System.out.print(i + ". " + functions.get(i-1).toString() + "     ");
                }

        }

        public void printProblem()
        {
                System.out.println("Problem Set Functions :");
                printAllFunctions(growthRateFunctions);
        }

        public void sortFunctionsByGrowthRatesAndPrint()
        {
                ArrayList<FunctionX> sortedGrowthRateFunctions = new
ArrayList<FunctionX>(growthRateFunctions);
                Collections.copy(sortedGrowthRateFunctions, growthRateFunctions);

                System.out.println("\n\nFunctions listed in order of asymptotic growth rates
(increasing):");
                sortFunctionsByGrowthRates(sortedGrowthRateFunctions);
                printAllFunctions(sortedGrowthRateFunctions);
        }

        public void sortFunctionsByGrowthRates(ArrayList<FunctionX> sortedGrowthRateFunctions)
        {
                Comparator<FunctionX> sortByGrowthRate = new GrowthRatesComparator();
                Collections.sort(sortedGrowthRateFunctions, sortByGrowthRate);

        }

        public void sortFunctionsByGrowthRates()
        {
                Comparator<FunctionX> sortByGrowthRate = new GrowthRatesComparator();
                Collections.sort(growthRateFunctions, sortByGrowthRate);
        }

        public String getSolvedStyledString()
        {
                StringBuilder stringBuilder = new StringBuilder();
                stringBuilder.append("Given functions:\n\n");
                for(int i = 0 ; i< originalFunctions.size() ; i++)
                {
                        int j = i+1;
                        stringBuilder.append("f"+ j +"(n) = [b" +
originalFunctions.get(i).toStyledString() + "]\n");
                }

                stringBuilder.append("\n");

                stringBuilder.append("The arrangement of the functions given in the problem\n");
                stringBuilder.append("i.e. f1(n), f2(n)..., f5(n) in ascending order of growth\n");
                stringBuilder.append("such that f1(n) = Omega(f2(n)),  f2(n) = Omega((f3(n)), ... f4(n) =
Omega(f5(n)) is :-\n\n");
```

```
                stringBuilder.append("[b[l");
                for (FunctionX fx : growthRateFunctions)
        {
                stringBuilder.append(fx.toStyledString() + " < ");
                }
        stringBuilder = stringBuilder.delete(stringBuilder.length() - 2, stringBuilder.length());
        stringBuilder.append("]]");

        return stringBuilder.toString();
        }

}
```

File: CaseOfTheorem .java

```
package algorithmLibrary;

public enum CaseOfTheorem {
        case1,
        case2,
        case3,
        notsolvableByMasterTheorem
}
```

File: ComplexityType.java

```
package algorithmLibrary;

public enum ComplexityType {

                BigO,
                Omega,
                Theta;

                public String toStyledString()
                {
                        return name();
                }
}
```

File: Constant.java

```
package algorithmLibrary;


public class Constant extends Operation
{
```

```
	Double value;
	public Constant(double p_value)
	{
		value = new Double(p_value);
	}

	public String toString()
	{
		String result ="";
		result+= str(value);
		return result;

	}

	@Override
	public String toStyledString() {
		String result ="";
		result+= str(value);
		return result;

	}

	public String str(Double var)
	{
		if(Math.floor(var) == var)
		{
			Integer varInt = var.intValue();
			return varInt.toString();
		}

		return var.toString();
	}

	public double value(double n)
	{
		return value;
	}


}
```

File: Divide.java

```
package algorithmLibrary;

public class Divide extends Operation{

	Operation numerator;
	Operation divideby;


	public Divide(Operation p_numerator, Operation p_divideby)
	{
```

```
		numerator = p_numerator;
		divideby = p_divideby;
	}

	public Divide(Operation p_numerator, double p_divideby)
	{
		numerator = p_numerator;
		divideby = new Constant(p_divideby);
	}

	public String toString()
	{
		String result ="";
		result+= "(" + numerator.toString() + "/" + divideby.toString() + ")";
		return result;
	}

	@Override
	public String toStyledString() {
		String result ="";
		result+= "(" + numerator.toString() + "/" + divideby.toString() + ")";
		return result;
	}


	@Override
	public double value(double n)
	{
		return numerator.value(n)/divideby.value(n);
	}
}
```

File: ExpToN.java

```
package algorithmLibrary;

public class ExpToN extends Operation
{
	Operation base;
	public ExpToN(double p_base)
	{
		base = new Constant(p_base);
	}

	public ExpToN(Operation p_base)
	{
		base = p_base;
	}

	public String toString()
	{
		String result ="";

		if(base.toString().equals("1"))
```

```
		{
			result+= "";
		}
		else if(base.toString().equals("0"))
		{
			result+= "0";
		}
		else
		{
			result+= "" + base.toString()+ "^n";
		}

		return result;

	}

	@Override
	public String toStyledString() {
		String result ="";

		if(base.toString().equals("1"))
		{
			result+= "";
		}
		else if(base.toString().equals("0"))
		{
			result+= "0";
		}
		else
		{
			result+= "(" + base.toString()+ "[un])";
		}

		return result;
	}



	public double value(double n)
	{
		return Math.pow(base.value(n), n);
	}
}
```

File: ExpToOperand.java

```
package algorithmLibrary;

public class ExpToOperand extends Operation{

	Operation base;
	Operation power;
```

```
public ExpToOperand(Operation p_base, Operation p_power)
{
	base = p_base;
	power = p_power;
}

public ExpToOperand(double p_base, Operation p_power)
{
	base = new Constant(p_base);
	power = p_power;
}

public String toString()
{
	String result ="";

	if(base.toString().equals("1"))
	{
		result+= "1";
	}
	else if(base.toString().equals("0"))
	{
		result+= "0";
	}
	else
	{
		result+= "(" + base.toString() + "^" + power.toString() + ")";
	}

	return result;

}



@Override
public double value(double n)
{
	return Math.pow(base.value(n), power.value(n));
}

@Override
public String toStyledString() {
	String result ="";

	if(base.toString().equals("1"))
	{
		result+= "1";
	}
	else if(base.toString().equals("0"))
	{
		result+= "0";
	}
	else
	{
		result+= "" + base.toString() + "[u" + power.toString() + "]";
```

```
		}

		return result;

	}

}
```

File: FactorialN.java

```
package algorithmLibrary;


public class FactorialN extends Operation
{

	public String toString()
	{
		String result ="";
		result+= "n!";
		return result;

	}


	public double value(double n)
	{

		// as n! grows way too fast at even numbers like n = 100 , stack over occurs
		// because of recursive stacks, therefore making non recursive code

		double returnValue = 1.0;
		for(double i = 1 ; i <= n ; i++)
		{
			returnValue *= i;
		}

		return returnValue;


	}


	@Override
	public String toStyledString() {
		String result ="";
		result+= "n!";
		return result;

	}

}
```

File: FactorialOfOperand.java

```
package algorithmLibrary;
```

```
public class FactorialOfOperand extends Operation{

	Operation operand;

	public FactorialOfOperand(Operation p_operand)
	{
		operand = p_operand;
	}

	public String toString()
	{
		String result ="";
		result+= operand.toString() + "!";
		return result;

	}
	@Override
	public double value(double n)
	{

		// as n! grows way too fast at even numbers like n = 100 , stack over occurs
		// because of recursive stacks, therefore making non recursive code

		double returnValue = 1.0;
		for(double i = 1 ; i <= operand.value(n) ; i++)
		{
			returnValue *= i;
		}

		return returnValue;


	}

	@Override
	public String toStyledString() {
		String result ="";
		result+= operand.toString() + "!";
		return result;
	}

}
```

File: FunctionX.java

```
package algorithmLibrary;
import java.util.ArrayList;


public class FunctionX extends Operation
{
	public Sign signMain;
	public ArrayList<Operation> terms;
```

```
public Sign signConnection;
public ArrayList<Operation> secondterms;
public Sign signConstant;
public ArrayList<Operation> constantTerms;



Double coefficient;


public FunctionX()
{
	terms = new ArrayList<Operation>();
	secondterms = new ArrayList<Operation>();
	constantTerms = new ArrayList<Operation>();
	signMain = new Sign("+");
	signConnection = new Sign("-");
	signConstant = new Sign("-");
}

public void addTerm(Operation termobj)
{
	terms.add(termobj);
}

public void addSecondTerm(Operation termobj)
{
	secondterms.add(termobj);
}

public void addSecondTerm(Operation termobj, String sign)
{
	secondterms.add(termobj);
	signConnection = new Sign(sign);
}

public void addConstant(Operation termobj)
{
	constantTerms.add(termobj);
}

public void addConstant(Operation termobj, String sign)
{
	constantTerms.add(termobj);
	signConstant = new Sign(sign);
}

public void setConnectionSign(String sign)
{
	signConnection = new Sign(sign);
}

public void setMainSign(String sign)
{
	signMain = new Sign(sign);
}
```

```
public void setConstantSign(String sign)
{
	signConstant = new Sign(sign);
}

public String toString()
{
	String result = "";

	for(Operation obj : terms)
	{
		if(!obj.toString().equals("1"))
		{
			result = result + obj.toString() +" ";
		}
	}

	result = result.trim();

	if(!result.equals(""))
		return result;
	else
		return "1";

}


public double value(double n)
{
	double returnValue = 1;

	for(Operation obj : terms)
	{
		returnValue *= obj.value(n);
	}

	if(!signMain.ispositive)
	{
		returnValue = returnValue * -1.0;
	}


	//secondTerms support
	if(secondterms.size() > 0)
	{

		double secondTermsValue = 1;
		for(Operation obj : secondterms)
		{
			secondTermsValue *= obj.value(n);
		}

		if(signConnection.ispositive)
		{
			returnValue = returnValue + secondTermsValue;
```

```
                }
                else
                {
                        returnValue = returnValue - secondTermsValue;
                }
        }

        //constant support
        if(constantTerms.size() > 0)
        {

                double constantTermsValue = 1;
                for(Operation obj : constantTerms)
                {
                        constantTermsValue *= obj.value(n);
                }

                if(signConstant.ispositive)
                {
                        returnValue = returnValue + constantTermsValue;
                }
                else
                {
                        returnValue = returnValue - constantTermsValue;
                }
        }

        return returnValue;
}


public String toStyledString()
{
        String result = "";

        if(!signMain.ispositive){
                result = result + " " + signMain.toStyledString() + " ";
        }

        for(Operation obj : terms)
        {
                if(!obj.toStyledString().equals("1"))
                {
                        result = result + obj.toStyledString() +" ";
                }
        }

        result = result.trim();

        //secondTerms support

        if(secondterms.size() > 0)
        {
                result = result + " " + signConnection.toStyledString() + " ";

                for(Operation obj : secondterms)
```

```
                        {
                                if(!obj.toStyledString().equals("1"))
                                {
                                        result = result + obj.toStyledString() +" ";
                                }
                        }
                }

                result = result.trim();

                //constant support

                if(constantTerms.size() > 0)
                {
                        result = result + " " + signConstant.toStyledString() + " ";

                        for(Operation obj : constantTerms)
                        {
                                result = result + obj.toStyledString() +" ";
                        }
                }

                result = result.trim();

                if(!result.equals(""))
                        return result;
                else
                        return "1";

        }


        public ArrayList<Operation> getTerms() {
                return terms;
        }

        public ArrayList<Operation> getSecondTerms() {
                return secondterms;
        }

}
```

File: GrowthRatesComparator.java

```
package algorithmLibrary;

import java.util.Comparator;

public class GrowthRatesComparator  implements Comparator<FunctionX>{

        enum caseOfComparison
        {
                AGreater,
                BGreater,
```

```
		ABEqual
	}

	@Override
	public int compare(FunctionX function_a, FunctionX function_b) {

		double aVal = 0.0f;
		double bVal = 0.0f;
		double startingNValue = 10.0;
		double maxOrderOfCheck = 10;
		double orderOfGrowth = 100;
		caseOfComparison whichCase = caseOfComparison.ABEqual;

		// comparing the values of the functions for various orders of n
		// might have to change the logic for so many comparisons
		// because anyways the checking is happening on the biggest value of n

		long countAGreater = 0;
		long countBGreater = 0;
		long countABEqual = 0;


		for(double n= startingNValue , i = 1; i <= maxOrderOfCheck ; n = n * orderOfGrowth ,
i++)
		{
			aVal = function_a.value(n);
			bVal = function_b.value(n);

			if(aVal > bVal)
			{
				whichCase = caseOfComparison.AGreater;
				countAGreater++;
			}
			else if(bVal > aVal)
			{
				whichCase = caseOfComparison.BGreater;
				countBGreater++;
			}
			else
			{
				whichCase = caseOfComparison.ABEqual;
				countABEqual++;
			}

			// to avoid computation to extremely huge numbers
			if(aVal >= Math.pow(Double.MAX_VALUE, .1)  ||   aVal >=
Math.pow(Double.MAX_VALUE, .1) )
				break;


		}

		if(whichCase == caseOfComparison.AGreater)
		{
			return 1;
		}
```

```
                else if(whichCase == caseOfComparison.BGreater)
                {
                        return -1;
                }
                else
                {
                        return 0;
                }

        }
}
```

File: LogN.java

```
package algorithmLibrary;
public class LogN extends Operation
{
        Operation base;
        public LogN(double p_base)
        {
                base = new Constant(p_base);
        }

        public LogN(Operation p_base)
        {
                base = p_base;
        }

        public String toString()
        {
                String result ="";
                if(base.toString().equals("2"))
                {
                        result+= "(lg n)";
                }
                else
                {
                        result+= "(log b-" + base.toString() + " n)";
                }
                return result;

        }



        public double value(double n)
        {
                return ( Math.log(n)/ Math.log(base.value(n)) );
        }

        @Override
        public String toStyledString() {
                String result ="";
                if(base.toString().equals("2"))
```

```
                {
                        result+= "(lg n)";
                }
                else
                {
                        result+= "(log[d" + base.toString() + "] n)";
                }
                return result;

        }


}
```

File: LogOfOperand.java

```
package algorithmLibrary;
public class LogOfOperand extends Operation
{
        Operation base;
        Operation operand;

   public LogOfOperand(Operation p_base, Operation p_operand)
        {
                base = p_base;
                operand = p_operand;
        }

   public LogOfOperand(double p_base, Operation p_operand)
        {
                base = new Constant(p_base);
                operand = p_operand;
        }
        public String toString()
        {
                String result ="";
                if(base.toString().equals("2"))
                {
                        result+= "(lg " + operand.toString() + ")";
                }
                else
                {
                        result+= "(log b-" + base.toString() + " "+ operand.toString() + ")";
                }
                return result;

        }



        public double value(double n)
        {
                return ( Math.log(operand.value(n))/ Math.log(base.value(n)) );
        }
```

```
	@Override
	public String toStyledString() {
		String result ="";
		if(base.toString().equals("2"))
		{
			result+= "(lg " + operand.toString() + ")";
		}
		else
		{
			result+= "(log[d" + base.toString() + "] "+ operand.toString() + ")";
		}
		return result;

	}


}
```

File: MasterTheorem.java

```
package algorithmLibrary;

import java.util.ArrayList;

public class MasterTheorem
{
	Recurrence recurrence;


	public MasterTheorem(Recurrence p_recurrence)
	{
		recurrence = new Recurrence();
		recurrence = p_recurrence;
	}

	public ArrayList<Object> getComputedResultValues()
	{
		ArrayList<Object> resultValues = new ArrayList<Object>();

		//a
		resultValues.add(recurrence.a);

		//b
		resultValues.add(recurrence.b);

		//nLogba
		resultValues.add(logabValCompute());


	CaseOfTheorem caseObj = whichCase();

		if(caseObj == CaseOfTheorem.case1)
		{
			// case1
```

```
                    resultValues.add("O");
                    resultValues.add("-e");
                    resultValues.add("Case1");
                }
                else if(caseObj == CaseOfTheorem.case2)
                {
                    // case2
                    resultValues.add("Theta");
                    resultValues.add("");
                    resultValues.add("Case2");
                }
                else if(caseObj == CaseOfTheorem.case3)
                {
                    // case3
                    resultValues.add("Omega");
                    resultValues.add("+e");
                    resultValues.add("Case3");
                }


                return resultValues;
        }


        public void solve()
        {
                CaseOfTheorem caseObj = whichCase();

                if(caseObj == CaseOfTheorem.case1)
                {
                    // case1

                    // used for computing the bound for n^log b-b a
                    Operation NLogbaBigThetaTerm = new NPower(logabValCompute());
                    FunctionX NLogbaBigThetaTermFunction = new FunctionX();
                    NLogbaBigThetaTermFunction.addTerm(NLogbaBigThetaTerm);

                    String nExplogabValComputeString = "";
                    if(logabValCompute() == 1.0)
                    {
                        nExplogabValComputeString = "n";
                    }
                    else if(logabValCompute() == 0.0)
                    {
                        nExplogabValComputeString = "";
                    }
                    else
                    {
                        nExplogabValComputeString = "n^" + str(logabValCompute()) ;
                    }
                    System.out.println("\nSolution by Master Method");
                    System.out.println("\nFor this recurrence, we have a = " + str(recurrence.a) + ",
b = " + str(recurrence.b) + ", f(n) = " + recurrence.fx.toString() + ",");
                    System.out.println("and thus n^(log b-b of a) = n^(log b-" + str(recurrence.b) + "
" + str(recurrence.a) + ") = Big Theta(" + NLogbaBigThetaTermFunction.toString() + ").");
```

```
                        System.out.println("Since f(n) = Big O(n^(log b-" + str(recurrence.b)+ " " +
str(recurrence.a) + " - e)), where e is some constant > 0," );
                        System.out.println("we can apply case 1 of the Master Theorem and conclude
that");
                        System.out.println("T(n) = Big Theta(" + nExplogabValComputeString + ")" + "
or  " + "T(n) = Big Theta(n^log b-" + str(recurrence.b) + " " + str(recurrence.a) + ")");

                }
                else if(caseObj == CaseOfTheorem.case2)
                {
                        // case2

                        // used for computing the bound for n^log b-b a
                        Operation NLogbaBigThetaTerm = new NPower(logabValCompute());
                        FunctionX NLogbaBigThetaTermFunction = new FunctionX();
                        NLogbaBigThetaTermFunction.addTerm(NLogbaBigThetaTerm);

                        String nExplogabValComputeString = "";
                        if(logabValCompute() == 1.0)
                        {
                                nExplogabValComputeString = "n ";
                        }
                        else if(logabValCompute() == 0.0)
                        {
                                nExplogabValComputeString = "";
                        }
                        else
                        {
                                nExplogabValComputeString = "n^" + str(logabValCompute()) +" " ;
                        }

                        System.out.println("\nSolution by Master Method");
                        System.out.println("\nFor this recurrence, we have a = " + str(recurrence.a) + ",
b = " + str(recurrence.b) + ", f(n) = " + recurrence.fx.toString() + ",");
                        System.out.println("and thus n^(log b-b of a) = n^(log b-" + str(recurrence.b) + "
" + str(recurrence.a) + ") = Big Theta(" + NLogbaBigThetaTermFunction.toString() + ").");
                        System.out.println("Since f(n) = Big Theta(n^(log b-" + str(recurrence.b)+ " " +
str(recurrence.a) + ")),");
                        System.out.println("we can apply case 2 of the Master Theorem and conclude
that");
                        System.out.println("T(n) = Big Theta(" + nExplogabValComputeString + "lg n)"
+ "  or  " + "T(n) = Big Theta(n^log b-" + str(recurrence.b) + " " + str(recurrence.a) + " lg n)");

                }
                else if(caseObj == CaseOfTheorem.case3)
                {
                        // case3

                        // used for computing the bound for n^log b-b a
                        Operation NLogbaBigThetaTerm = new NPower(logabValCompute());
                        FunctionX NLogbaBigThetaTermFunction = new FunctionX();
                        NLogbaBigThetaTermFunction.addTerm(NLogbaBigThetaTerm);


                        System.out.println("\nSolution by Master Method");
```

```
                        System.out.println("\nFor this recurrence, we have a = " + str(recurrence.a) + ",
b = " + str(recurrence.b) + ", f(n) = " + recurrence.fx.toString() + ",");
                        System.out.println("and thus n^(log b-b of a) = n^(log b-" + str(recurrence.b) + "
" + str(recurrence.a) + ") = Big Theta(" + NLogbaBigThetaTermFunction.toString() + ").");
                        System.out.println("Since f(n) = Big Omega(n^(log b-" + str(recurrence.b)+ " "
+ str(recurrence.a) + " + e)), where e is some constant > 0," );
                        System.out.println("We can apply case 3 of the Master Theorem and conclude
that");
                        System.out.println("T(n) = Big Theta(" + recurrence.fx.toString() +  ")");

                }
                else
                {
                        // notsolvableByMasterTheorem

                        System.out.println("\nSolution by Master Method");
                        System.out.println("\nWe can not apply the master method because the
recurrence falls");
                        System.out.println("into the gap between either Case 1 and Case 2 or the gap
between Case 2 and Case 3");

                }

        }



        public String getSolvedStyledString()
        {
                StringBuffer resultStr = new StringBuffer();

    CaseOfTheorem caseObj = whichCase();

                if(caseObj == CaseOfTheorem.case1)
                {
                        // case1

                        // used for computing the bound for n^log b-b a
                        Operation NLogbaBigThetaTerm = new
NPower(roundToDecimals(logabValCompute(),2));
                        FunctionX NLogbaBigThetaTermFunction = new FunctionX();
                        NLogbaBigThetaTermFunction.addTerm(NLogbaBigThetaTerm);

                        String nExplogabValComputeString = "";
                        if(logabValCompute() == 1.0)
                        {
                                nExplogabValComputeString = "n";
                        }
                        else if(logabValCompute() == 0.0)
                        {
                                nExplogabValComputeString = "";
                        }
                        else
                        {
                                nExplogabValComputeString = "n[u" +
str(roundToDecimals(logabValCompute(),2))+ "]" ;
```

```
                        }
                        resultStr.append("[b[lSolution by Master Theorem - ]]");
                        resultStr.append("\n\nFor this recurrence, we have \n\n[ba = " + str(recurrence.a)
+ "], \n[bb = " + str(recurrence.b) + "], \n[bf(n) = " + recurrence.fx.toStyledString() + "],");
                        resultStr.append("\n\nand thus \n[b[d[xn]](log [db] a) = [d[xn]](log b[d" +
str(recurrence.b) + "] " + str(recurrence.a) + ") \n= Big Theta(" +
NLogbaBigThetaTermFunction.toStyledString() + ")].");
                        resultStr.append("\n\n[d[xSince [bf(n) = Big O(n)]][b(log b[d" +
str(recurrence.b)+ "] " + str(recurrence.a) + " - e)])], \nwhere e is some constant > 0," );
                        resultStr.append("\n\nwe can apply [b[ccase 1]] of the Master Theorem and
conclude that");
                        resultStr.append("\n\n[d[x[b[cT(n) = Big Theta(n)]]][b[l[clog [d" +
str(recurrence.b) + "] " + str(recurrence.a) + "])]]] or " + "\n\n[b[l[cT(n) = Big Theta(" +
nExplogabValComputeString + ")" + "]]]");

                }
                else if(caseObj == CaseOfTheorem.case2)
                {
                        // case2

                        // used for computing the bound for n^log b-b a
                        Operation NLogbaBigThetaTerm = new
NPower(roundToDecimals(logabValCompute(),2));
                        FunctionX NLogbaBigThetaTermFunction = new FunctionX();
                        NLogbaBigThetaTermFunction.addTerm(NLogbaBigThetaTerm);

                        String nExplogabValComputeString = "";
                        if(logabValCompute() == 1.0)
                        {
                                nExplogabValComputeString = "n ";
                        }
                        else if(logabValCompute() == 0.0)
                        {
                                nExplogabValComputeString = "";
                        }
                        else
                        {
                                nExplogabValComputeString = "n[u" +
str(roundToDecimals(logabValCompute(),2))+ "]" ;
                        }

                        resultStr.append("[b[lSolution by Master Theorem - ]]");
                        resultStr.append("\n\nFor this recurrence, we have \n\n[ba = " + str(recurrence.a)
+ "], \n[bb = " + str(recurrence.b) + "], \n[bf(n) = " + recurrence.fx.toStyledString() + "],");
                        resultStr.append("\n\nand thus \n[b[d[xn]](log [db] a) = [d[xn]](log [d" +
str(recurrence.b) + "] " + str(recurrence.a) + ") \n= Big Theta(" +
NLogbaBigThetaTermFunction.toStyledString() + ")].");
                        resultStr.append("\n\n[d[xSince [bf(n) = Big Theta(n)]][b(log [d" +
str(recurrence.b)+ "] " + str(recurrence.a) + ")])]," );
                        resultStr.append("\n\nwe can apply [b[ccase 2]] of the Master Theorem and
conclude that");
                        resultStr.append("\n\n[d[x[b[cT(n) = Big Theta(n)]]][b[l[clog [d" +
str(recurrence.b) + "] " + str(recurrence.a) + "] lg n)]]]"  + "  or  " + "\n\n[b[l[cT(n) = Big Theta(" +
nExplogabValComputeString + "lg n)]]]");

                }
```

```
		else if(caseObj == CaseOfTheorem.case3)
		{
			// case3

			// used for computing the bound for n^log b-b a
			Operation NLogbaBigThetaTerm = new
NPower(roundToDecimals(logabValCompute(),2));
			FunctionX NLogbaBigThetaTermFunction = new FunctionX();
			NLogbaBigThetaTermFunction.addTerm(NLogbaBigThetaTerm);


			resultStr.append("[b[lSolution by Master Theorem - ]]");
			resultStr.append("\n\nFor this recurrence, we have \n\n[ba = " + str(recurrence.a)
+ "], \n[bb = " + str(recurrence.b) + "], \n[bf(n) = " + recurrence.fx.toStyledString() + "],");
			resultStr.append("\n\nand thus [b[d[xn]](log [db] a) = [dn](log [d" +
str(recurrence.b) + "] " + str(recurrence.a) + ") \n= Big Theta(" +
NLogbaBigThetaTermFunction.toStyledString() + ")].");
			resultStr.append("\n\n[d[xSince [bf(n) = Big Omega(n)]][b(log [d" +
str(recurrence.b)+ "] " + str(recurrence.a) + " + e)])], \nwhere e is some constant > 0," );
			resultStr.append("\n\nWe can apply [b[ccase 3]] of the Master Theorem and
conclude that");
			resultStr.append("\n\n[b[l[cT(n) = Big Theta(" + recurrence.fx.toStyledString()
+  ")]]]");

		}
		else
		{
			// notsolvableByMasterTheorem

			resultStr.append("[b[lSolution by Master Theorem - ]]");
			resultStr.append("\n\nWe can not apply the master theorem because the
recurrence falls");
			resultStr.append("into the gap between either Case 1 and Case 2 or the gap
between Case 2 and Case 3.");

		}

		return new String(resultStr);
	}

	public TimeComplexityMasterMethod getSolvedBound()
	{
		TimeComplexity timeComplexity;
		CaseOfTheorem solvedCase = whichCase();

		Operation operation;
		FunctionX fx;

		switch (solvedCase) {
		case case1:

			operation = new NPower(new LogOfOperand(recurrence.b, new
Constant(recurrence.a)));
			fx = new FunctionX();
			fx.addTerm(operation);
			timeComplexity = new TimeComplexity(ComplexityType.Theta , fx);
```

```
                        break;

        case case2:

            operation = new NPower(new LogOfOperand(recurrence.b, new Constant(recurrence.a)));
                        fx = new FunctionX();
                        fx.addTerm(operation);
                        fx.addTerm(new LogN(2));
                        timeComplexity = new TimeComplexity(ComplexityType.Theta , fx);

                        break;

        case case3:

            fx = recurrence.fx;
                        timeComplexity = new TimeComplexity(ComplexityType.Theta , fx);

                break;

        case notsolvableByMasterTheorem:

            //in case the Master Method can not solve the recurrence
            //constant -1.0 is passed back as time complexity fx
            fx = new FunctionX();
            fx.addTerm(new Constant(-1.0));
            timeComplexity = new TimeComplexity(ComplexityType.Theta , fx);
                break;

        default:
            timeComplexity = new TimeComplexity();
                        break;
                }


                return new TimeComplexityMasterMethod(solvedCase,
timeComplexity.getComplexityType(), timeComplexity.getFunctionX());
            }



            public TimeComplexityMasterMethod getRoundedSolvedBound()
            {
                TimeComplexity timeComplexity;
                CaseOfTheorem solvedCase = whichCase();

                Operation operation;
                FunctionX fx;

                switch (solvedCase) {
                case case1:

                        operation = new NPower(Math.rint
((Math.log(recurrence.b)/Math.log(recurrence.a))    ));
                        fx = new FunctionX();
                        fx.addTerm(operation);
```

```
                        timeComplexity = new TimeComplexity(ComplexityType.Theta , fx);

                        break;

    case case2:

        operation = new NPower(Math.rint   ((Math.log(recurrence.b)/Math.log(recurrence.a))    ));
                        fx = new FunctionX();
                        fx.addTerm(operation);
                        fx.addTerm(new LogN(2));
                        timeComplexity = new TimeComplexity(ComplexityType.Theta , fx);

                        break;

    case case3:

        fx = recurrence.fx;
                        timeComplexity = new TimeComplexity(ComplexityType.Theta , fx);

            break;

    case notsolvableByMasterTheorem:

        //in case the Master Method can not solve the recurrence
        //constant -1.0 is passed back as time complexity fx
        fx = new FunctionX();
        fx.addTerm(new Constant(-1.0));
        timeComplexity = new TimeComplexity(ComplexityType.Theta , fx);
            break;

    default:
        timeComplexity = new TimeComplexity();
                        break;
                }


                return new TimeComplexityMasterMethod(solvedCase,
timeComplexity.getComplexityType(), timeComplexity.getFunctionX());
        }


        public CaseOfTheorem whichCase()
        {

                double fxVal = 0.0f;
                double nExpLogbaVal = 0.0f;
                double maxOrderOfCheck = 100;
                double orderOfGrowth = 10;
                //checking for case 1
                //if n^logb a  is polynomially greater than fx


                // will have to change strategy for determining
                // which function is polynomially greater

                long countCase1Greater = 0;
```

```
long  countCase1Lesser = 0;
boolean case1 = false;


for(double n= 1, i = 1; i <= maxOrderOfCheck ; n = n * orderOfGrowth , i++)
{
	nExpLogbaVal = nExpLogabValCompute(n);
	fxVal = recurrence.fx.value(n);

	if(nExpLogbaVal > fxVal)
	{
		case1 = true;
		countCase1Greater++;
	}
	else
	{
		case1 = false;
		countCase1Lesser++;
	}

	// to avoid computation to extremely huge numbers
	if(fxVal >= Math.sqrt(Double.MAX_VALUE))
		break;


}

if(case1 == true)return CaseOfTheorem.case1;



// reinitializing the variables to 0
fxVal = 0.0f;
nExpLogbaVal = 0.0f;

//checking for case 2
//if n^logb a  is equal than fx


long countCase2Equal = 0;
long countCase2NotEqual = 0;
boolean case2 = false;



for(double n= 1, i = 1; i <= maxOrderOfCheck ; n = n * orderOfGrowth , i++)
{
	nExpLogbaVal = nExpLogabValCompute(n);
	fxVal = recurrence.fx.value(n);

	if(nExpLogbaVal == fxVal)
	{
		case2 = true;
		countCase2Equal++;
	}
	else
```

```
		{
			case2 = false;
			countCase2NotEqual++;
		}

		// to avoid computation to extremely huge numbers
		if(fxVal >= Math.sqrt(Double.MAX_VALUE))
			break;
}

if(case2 == true)return CaseOfTheorem.case2;



//checking for case 3
//if n^logb a  is polynomially lesser than fx


// will have to change strategy for determining
// which function is polynomially greater
// and also check the regularity -
// af(n/b) <= cf(n) for some c<1 and all sufficiently large n

long countCase3Greater = 0;
long  countCase3Lesser = 0;
boolean case3 = false;



for(double n= 1, i = 1; i <= maxOrderOfCheck ; n = n * orderOfGrowth , i++)
{
		nExpLogbaVal = nExpLogabValCompute(n);
		fxVal = recurrence.fx.value(n);

		if(nExpLogbaVal < fxVal)
		{
			case3 = true;
			countCase3Lesser++;
		}
		else
		{
			case3 = false;
			countCase3Greater++;
		}

		// to avoid computation to extremely huge numbers
		if(fxVal >= Math.sqrt(Double.MAX_VALUE))
			break;
}

if(case3 == true)return CaseOfTheorem.case3;



// if the comparisons of the functions does not lie on any
// of the above cases
```

```
                return CaseOfTheorem.notsolvableByMasterTheorem;

        }


        double nExpLogabValCompute(double n)
        {
                double logba = Math.log(recurrence.a) / Math.log(recurrence.b);
                return Math.pow(n, logba);
        }

        double logabValCompute()
        {
                return Math.log(recurrence.a) / Math.log(recurrence.b);
        }

        public static double roundToDecimals(double d, int c)
        {
                int temp=(int)((d*Math.pow(10,c)));
                return (((double)temp)/Math.pow(10,c));
        }

        public String str(Double var)
        {
                if(Math.floor(var) == var)
                {
                        Integer varInt = var.intValue();
                        return varInt.toString();
                }

                return var.toString();
        }

}



package algorithmLibrary;

import java.util.ArrayList;

public class MasterTheorem
{
        Recurrence recurrence;


        public MasterTheorem(Recurrence p_recurrence)
        {
                recurrence = new Recurrence();
                recurrence = p_recurrence;
        }

        public ArrayList<Object> getComputedResultValues()
        {
                ArrayList<Object> resultValues = new ArrayList<Object>();
```

```
		//a
		resultValues.add(recurrence.a);

		//b
		resultValues.add(recurrence.b);

		//nLogba
		resultValues.add(logabValCompute());


CaseOfTheorem caseObj = whichCase();

		if(caseObj == CaseOfTheorem.case1)
		{
			// case1
			resultValues.add("O");
			resultValues.add("-e");
			resultValues.add("Case1");
		}
		else if(caseObj == CaseOfTheorem.case2)
		{
			// case2
			resultValues.add("Theta");
			resultValues.add("");
			resultValues.add("Case2");
		}
		else if(caseObj == CaseOfTheorem.case3)
		{
			// case3
			resultValues.add("Omega");
			resultValues.add("+e");
			resultValues.add("Case3");
		}


		return resultValues;
	}


	public void solve()
	{
		CaseOfTheorem caseObj = whichCase();

		if(caseObj == CaseOfTheorem.case1)
		{
			// case1

			// used for computing the bound for n^log b-b a
			Operation NLogbaBigThetaTerm = new NPower(logabValCompute());
			FunctionX NLogbaBigThetaTermFunction = new FunctionX();
			NLogbaBigThetaTermFunction.addTerm(NLogbaBigThetaTerm);

			String nExplogabValComputeString = "";
			if(logabValCompute() == 1.0)
			{
				nExplogabValComputeString = "n";
```

```
                        }
                        else if(logabValCompute() == 0.0)
                        {
                                nExplogabValComputeString = "";
                        }
                        else
                        {
                                nExplogabValComputeString = "n^" + str(logabValCompute()) ;
                        }
                        System.out.println("\nSolution by Master Method");
                        System.out.println("\nFor this recurrence, we have a = " + str(recurrence.a) + ",
b = " + str(recurrence.b) + ", f(n) = " + recurrence.fx.toString() + ",");
                        System.out.println("and thus n^(log b-b of a) = n^(log b-" + str(recurrence.b) + "
" + str(recurrence.a) + ") = Big Theta(" + NLogbaBigThetaTermFunction.toString() + ").");
                        System.out.println("Since f(n) = Big O(n^(log b-" + str(recurrence.b)+ " " +
str(recurrence.a) + " - e)), where e is some constant > 0," );
                        System.out.println("we can apply case 1 of the Master Theorem and conclude
that");
                        System.out.println("T(n) = Big Theta(" + nExplogabValComputeString + ")" + "
or  " + "T(n) = Big Theta(n^log b-" + str(recurrence.b) + " " + str(recurrence.a) + ")");

                }
                else if(caseObj == CaseOfTheorem.case2)
                {
                        // case2

                        // used for computing the bound for n^log b-b a
                        Operation NLogbaBigThetaTerm = new NPower(logabValCompute());
                        FunctionX NLogbaBigThetaTermFunction = new FunctionX();
                        NLogbaBigThetaTermFunction.addTerm(NLogbaBigThetaTerm);

                        String nExplogabValComputeString = "";
                        if(logabValCompute() == 1.0)
                        {
                                nExplogabValComputeString = "n ";
                        }
                        else if(logabValCompute() == 0.0)
                        {
                                nExplogabValComputeString = "";
                        }
                        else
                        {
                                nExplogabValComputeString = "n^" + str(logabValCompute()) +" " ;
                        }

                        System.out.println("\nSolution by Master Method");
                        System.out.println("\nFor this recurrence, we have a = " + str(recurrence.a) + ",
b = " + str(recurrence.b) + ", f(n) = " + recurrence.fx.toString() + ",");
                        System.out.println("and thus n^(log b-b of a) = n^(log b-" + str(recurrence.b) + "
" + str(recurrence.a) + ") = Big Theta(" + NLogbaBigThetaTermFunction.toString() + ").");
                        System.out.println("Since f(n) = Big Theta(n^(log b-" + str(recurrence.b)+ " " +
str(recurrence.a) + ")),," );
                        System.out.println("we can apply case 2 of the Master Theorem and conclude
that");
                        System.out.println("T(n) = Big Theta(" + nExplogabValComputeString + "lg n)"
+ "  or  " + "T(n) = Big Theta(n^log b-" + str(recurrence.b) + " " + str(recurrence.a) + " lg n)");
```

```
                }
                else if(caseObj == CaseOfTheorem.case3)
                {
                        // case3

                        // used for computing the bound for n^log b-b a
                        Operation NLogbaBigThetaTerm = new NPower(logabValCompute());
                        FunctionX NLogbaBigThetaTermFunction = new FunctionX();
                        NLogbaBigThetaTermFunction.addTerm(NLogbaBigThetaTerm);


                        System.out.println("\nSolution by Master Method");
                        System.out.println("\nFor this recurrence, we have a = " + str(recurrence.a) + ",
b = " + str(recurrence.b) + ", f(n) = " + recurrence.fx.toString() + ",");
                        System.out.println("and thus n^(log b-b of a) = n^(log b-" + str(recurrence.b) + "
" + str(recurrence.a) + ") = Big Theta(" + NLogbaBigThetaTermFunction.toString() + ").");
                        System.out.println("Since f(n) = Big Omega(n^(log b-" + str(recurrence.b)+ " "
+ str(recurrence.a) + " + e)), where e is some constant > 0," );
                        System.out.println("We can apply case 3 of the Master Theorem and conclude
that");
                        System.out.println("T(n) = Big Theta(" + recurrence.fx.toString() +  ")");

                }
                else
                {
                        // notsolvableByMasterTheorem

                        System.out.println("\nSolution by Master Method");
                        System.out.println("\nWe can not apply the master method because the
recurrence falls");
                        System.out.println("into the gap between either Case 1 and Case 2 or the gap
between Case 2 and Case 3");

                }

        }



        public String getSolvedStyledString()
        {
                StringBuffer resultStr = new StringBuffer();

    CaseOfTheorem caseObj = whichCase();

                if(caseObj == CaseOfTheorem.case1)
                {
                        // case1

                        // used for computing the bound for n^log b-b a
                        Operation NLogbaBigThetaTerm = new
NPower(roundToDecimals(logabValCompute(),2));
                        FunctionX NLogbaBigThetaTermFunction = new FunctionX();
                        NLogbaBigThetaTermFunction.addTerm(NLogbaBigThetaTerm);
```

```
                        String nExplogabValComputeString = "";
                        if(logabValCompute() == 1.0)
                        {
                                nExplogabValComputeString = "n";
                        }
                        else if(logabValCompute() == 0.0)
                        {
                                nExplogabValComputeString = "";
                        }
                        else
                        {
                                nExplogabValComputeString = "n[u" +
str(roundToDecimals(logabValCompute(),2))+ "]" ;
                        }
                        resultStr.append("[b[lSolution by Master Theorem - ]]");
                        resultStr.append("\n\nFor this recurrence, we have \n\n[ba = " + str(recurrence.a)
+ "], \n[bb = " + str(recurrence.b) + "], \n[bf(n) = " + recurrence.fx.toStyledString() + "],");
                        resultStr.append("\n\nand thus \n[b[d[xn]](log [db] a) = [d[xn]](log b[d" +
str(recurrence.b) + "] " + str(recurrence.a) + ") \n= Big Theta(" +
NLogbaBigThetaTermFunction.toStyledString() + ")].");
                        resultStr.append("\n\n[d[xSince [bf(n) = Big O(n)]][b(log b[d" +
str(recurrence.b)+ "] " + str(recurrence.a) + " - e)])], \nwhere e is some constant > 0," );
                        resultStr.append("\n\nwe can apply [b[ccase 1]] of the Master Theorem and
conclude that");
                        resultStr.append("\n\n[d[x[b[cT(n) = Big Theta(n)]]][b[l[clog [d" +
str(recurrence.b) + "] " + str(recurrence.a) + "])]]] or " + "\n\n[b[l[cT(n) = Big Theta(" +
nExplogabValComputeString + ")" + "]]]");

                }
                else if(caseObj == CaseOfTheorem.case2)
                {
                        // case2

                        // used for computing the bound for n^log b-b a
                        Operation NLogbaBigThetaTerm = new
NPower(roundToDecimals(logabValCompute(),2));
                        FunctionX NLogbaBigThetaTermFunction = new FunctionX();
                        NLogbaBigThetaTermFunction.addTerm(NLogbaBigThetaTerm);

                        String nExplogabValComputeString = "";
                        if(logabValCompute() == 1.0)
                        {
                                nExplogabValComputeString = "n ";
                        }
                        else if(logabValCompute() == 0.0)
                        {
                                nExplogabValComputeString = "";
                        }
                        else
                        {
                                nExplogabValComputeString = "n[u" +
str(roundToDecimals(logabValCompute(),2))+ "]" ;
                        }

                        resultStr.append("[b[lSolution by Master Theorem - ]]");
```

```
                        resultStr.append("\n\nFor this recurrence, we have \n\n[ba = " + str(recurrence.a)
+ "], \n[bb = " + str(recurrence.b) + "], \n[bf(n) = " + recurrence.fx.toStyledString() + "],");
                        resultStr.append("\n\nand thus \n[b[d[xn]](log [db] a) = [d[xn]](log [d" +
str(recurrence.b) + "] " + str(recurrence.a) + ") \n= Big Theta(" +
NLogbaBigThetaTermFunction.toStyledString() + ")].");
                        resultStr.append("\n\n[d[xSince [bf(n) = Big Theta(n)]][b(log [d" +
str(recurrence.b)+ "] " + str(recurrence.a) + ")])]," );
                        resultStr.append("\n\nwe can apply [b[ccase 2]] of the Master Theorem and
conclude that");
                        resultStr.append("\n\n[d[x[b[cT(n) = Big Theta(n)]]][b[l[clog [d" +
str(recurrence.b) + "] " + str(recurrence.a) + "] lg n)]]]"  + "  or  " + "\n\n[b[l[cT(n) = Big Theta(" +
nExplogabValComputeString + "lg n)]]]");

                }
                else if(caseObj == CaseOfTheorem.case3)
                {
                        // case3

                        // used for computing the bound for n^log b-b a
                        Operation NLogbaBigThetaTerm = new
NPower(roundToDecimals(logabValCompute(),2));
                        FunctionX NLogbaBigThetaTermFunction = new FunctionX();
                        NLogbaBigThetaTermFunction.addTerm(NLogbaBigThetaTerm);


                        resultStr.append("[b[lSolution by Master Theorem - ]]");
                        resultStr.append("\n\nFor this recurrence, we have \n\n[ba = " + str(recurrence.a)
+ "], \n[bb = " + str(recurrence.b) + "], \n[bf(n) = " + recurrence.fx.toStyledString() + "],");
                        resultStr.append("\n\nand thus [b[d[xn]](log [db] a) = [dn](log [d" +
str(recurrence.b) + "] " + str(recurrence.a) + ") \n= Big Theta(" +
NLogbaBigThetaTermFunction.toStyledString() + ")].");
                        resultStr.append("\n\n[d[xSince [bf(n) = Big Omega(n)]][b(log [d" +
str(recurrence.b)+ "] " + str(recurrence.a) + " + e)])], \nwhere e is some constant > 0," );
                        resultStr.append("\n\nWe can apply [b[ccase 3]] of the Master Theorem and
conclude that");
                        resultStr.append("\n\n[b[l[cT(n) = Big Theta(" + recurrence.fx.toStyledString()
+  ")]]]");

                }
                else
                {
                        // notsolvableByMasterTheorem

                        resultStr.append("[b[lSolution by Master Theorem - ]]");
                        resultStr.append("\n\nWe can not apply the master theorem because the
recurrence falls");
                        resultStr.append("into the gap between either Case 1 and Case 2 or the gap
between Case 2 and Case 3.");

                }

                return new String(resultStr);
        }

        public TimeComplexityMasterMethod getSolvedBound()
        {
```

```
                TimeComplexity timeComplexity;
                CaseOfTheorem solvedCase = whichCase();

                Operation operation;
                FunctionX fx;

                switch (solvedCase) {
                case case1:

                        operation = new NPower(new LogOfOperand(recurrence.b, new
Constant(recurrence.a)));
                        fx = new FunctionX();
                        fx.addTerm(operation);
                        timeComplexity = new TimeComplexity(ComplexityType.Theta , fx);

                        break;

        case case2:

            operation = new NPower(new LogOfOperand(recurrence.b, new Constant(recurrence.a)));
                        fx = new FunctionX();
                        fx.addTerm(operation);
                        fx.addTerm(new LogN(2));
                        timeComplexity = new TimeComplexity(ComplexityType.Theta , fx);

                        break;

        case case3:

            fx = recurrence.fx;
                        timeComplexity = new TimeComplexity(ComplexityType.Theta , fx);

                break;

        case notsolvableByMasterTheorem:

            //in case the Master Method can not solve the recurrence
            //constant -1.0 is passed back as time complexity fx
            fx = new FunctionX();
            fx.addTerm(new Constant(-1.0));
            timeComplexity = new TimeComplexity(ComplexityType.Theta , fx);
                break;

        default:
            timeComplexity = new TimeComplexity();
                        break;
                }


                return new TimeComplexityMasterMethod(solvedCase,
timeComplexity.getComplexityType(), timeComplexity.getFunctionX());
        }



        public TimeComplexityMasterMethod getRoundedSolvedBound()
```

```
        {
                TimeComplexity timeComplexity;
                CaseOfTheorem solvedCase = whichCase();

                Operation operation;
                FunctionX fx;

                switch (solvedCase) {
                case case1:

                        operation = new NPower(Math.rint
((Math.log(recurrence.b)/Math.log(recurrence.a))    ));
                        fx = new FunctionX();
                        fx.addTerm(operation);
                        timeComplexity = new TimeComplexity(ComplexityType.Theta , fx);

                        break;

    case case2:

        operation = new NPower(Math.rint   ((Math.log(recurrence.b)/Math.log(recurrence.a))    ));
                        fx = new FunctionX();
                        fx.addTerm(operation);
                        fx.addTerm(new LogN(2));
                        timeComplexity = new TimeComplexity(ComplexityType.Theta , fx);

                        break;

    case case3:

        fx = recurrence.fx;
                        timeComplexity = new TimeComplexity(ComplexityType.Theta , fx);

            break;

    case notsolvableByMasterTheorem:

        //in case the Master Method can not solve the recurrence
        //constant -1.0 is passed back as time complexity fx
        fx = new FunctionX();
        fx.addTerm(new Constant(-1.0));
        timeComplexity = new TimeComplexity(ComplexityType.Theta , fx);
            break;

    default:
        timeComplexity = new TimeComplexity();
                        break;
                }


                return new TimeComplexityMasterMethod(solvedCase,
timeComplexity.getComplexityType(), timeComplexity.getFunctionX());
        }


        public CaseOfTheorem whichCase()
```

```
{

        double fxVal = 0.0f;
        double nExpLogbaVal = 0.0f;
        double maxOrderOfCheck = 100;
        double orderOfGrowth = 10;
        //checking for case 1
        //if n^logb a  is polynomially greater than fx


        // will have to change strategy for determining
        // which function is polynomially greater

        long countCase1Greater = 0;
        long  countCase1Lesser = 0;
        boolean case1 = false;


        for(double n= 1, i = 1; i <= maxOrderOfCheck ; n = n * orderOfGrowth , i++)
        {
                nExpLogbaVal = nExpLogabValCompute(n);
                fxVal = recurrence.fx.value(n);

                if(nExpLogbaVal > fxVal)
                {
                        case1 = true;
                        countCase1Greater++;
                }
                else
                {
                        case1 = false;
                        countCase1Lesser++;
                }

                // to avoid computation to extremely huge numbers
                if(fxVal >= Math.sqrt(Double.MAX_VALUE))
                        break;


        }

        if(case1 == true)return CaseOfTheorem.case1;



        // reinitializing the variables to 0
        fxVal = 0.0f;
        nExpLogbaVal = 0.0f;

        //checking for case 2
        //if n^logb a  is equal than fx


        long countCase2Equal = 0;
        long countCase2NotEqual = 0;
        boolean case2 = false;
```

```
for(double n= 1, i = 1; i <= maxOrderOfCheck ; n = n * orderOfGrowth , i++)
{
	nExpLogbaVal = nExpLogabValCompute(n);
	fxVal = recurrence.fx.value(n);

	if(nExpLogbaVal == fxVal)
	{
		case2 = true;
		countCase2Equal++;
	}
	else
	{
		case2 = false;
		countCase2NotEqual++;
	}

	// to avoid computation to extremely huge numbers
	if(fxVal >= Math.sqrt(Double.MAX_VALUE))
		break;
}

if(case2 == true)return CaseOfTheorem.case2;



//checking for case 3
//if n^logb a  is polynomially lesser than fx


// will have to change strategy for determining
// which function is polynomially greater
// and also check the regularity -
// af(n/b) <= cf(n) for some c<1 and all sufficiently large n

long countCase3Greater = 0;
long  countCase3Lesser = 0;
boolean case3 = false;



for(double n= 1, i = 1; i <= maxOrderOfCheck ; n = n * orderOfGrowth , i++)
{
	nExpLogbaVal = nExpLogabValCompute(n);
	fxVal = recurrence.fx.value(n);

	if(nExpLogbaVal < fxVal)
	{
		case3 = true;
		countCase3Lesser++;
	}
	else
	{
		case3 = false;
```

```
                        countCase3Greater++;
                }

                // to avoid computation to extremely huge numbers
                if(fxVal >= Math.sqrt(Double.MAX_VALUE))
                        break;
        }

        if(case3 == true)return CaseOfTheorem.case3;



        // if the comparisons of the functions does not lie on any
        // of the above cases
        return CaseOfTheorem.notsolvableByMasterTheorem;

    }


    double nExpLogabValCompute(double n)
    {
        double logba = Math.log(recurrence.a) / Math.log(recurrence.b);
        return Math.pow(n, logba);
    }

    double logabValCompute()
    {
        return Math.log(recurrence.a) / Math.log(recurrence.b);
    }

    public static double roundToDecimals(double d, int c)
    {
        int temp=(int)((d*Math.pow(10,c)));
        return (((double)temp)/Math.pow(10,c));
    }

    public String str(Double var)
    {
        if(Math.floor(var) == var)
        {
            Integer varInt = var.intValue();
            return varInt.toString();
        }

        return var.toString();
    }

}
```

File: NPower.java

```
package algorithmLibrary;
```

```
public class NPower extends Operation
{
	Operation power;
	public NPower(double p_power)
	{
		power = new Constant(p_power);
	}

	public NPower(Operation p_power)
	{
		power = p_power;
	}

	public String toString()
	{
		String result ="";

		if(power.toString().equals("1"))
		{
			result+= "n";
		}
		else if(power.toString().equals("0"))
		{
			result+= "1";
		}
		else
		{
			result+= "(n^" + power.toString() + ")";
		}

		return result;

	}


	public double value(double n)
	{
		return Math.pow(n, power.value(n));
	}

	@Override
	public String toStyledString() {

		String result ="";

		if(power.toString().equals("1"))
		{
			result+= "n";
		}
		else if(power.toString().equals("0"))
		{
			result+= "1";
		}
		else
		{
```

```
                result+= "n[u" + power.toString() + "]";
        }

        return result;

    }

}
```

File: OperandToPower.java

```
package algorithmLibrary;

public class OperandToPower extends Operation{


    Operation power;
    Operation base;


    public OperandToPower(Operation p_base, Operation p_power)
    {
        base = p_base;
        power = p_power;
    }

    public OperandToPower(Operation p_base, double p_power)
    {
        base = p_base;
        power = new Constant(p_power);
    }

    public String toString()
    {
        String result ="";

        if(power.toString().equals("1") )
        {
            result+= base.toString();
        }
        else if(base.toString().equals("1") || power.toString().equals("0") )
        {
            result+= "1";
        }
        else
        {
            result+= "(" + base.toString() + "^" + power.toString() + ")";
        }

        return result;

    }
```

```
	@Override
	public double value(double n)
	{
		return Math.pow(base.value(n), power.value(n));
	}

	@Override
	public String toStyledString() {

		String result ="";

		if(power.toString().equals("1") )
		{
			result+= base.toString();
		}
		else if(base.toString().equals("1") || power.toString().equals("0") )
		{
			result+= "1";
		}
		else
		{
			result+= "(" + base.toString() + "[u" + power.toString() + "])";
		}

		return result;

	}

}
```

File: Operation.java

```
package algorithmLibrary;
public abstract class Operation
{
	public abstract double value(double n);
	public abstract String toStyledString();
}
```

File: Recurrence.java

```
package algorithmLibrary;
public class Recurrence
{

	Double a = -1.0;
	Double b = -1.0;
```

```
public FunctionX fx;



public Recurrence()
{
	fx = new FunctionX();
}

public Recurrence(double p_a, double p_b, FunctionX p_fx)
{
	a = new Double(p_a);
	b = new Double(p_b);
	fx = p_fx;

}


public void PrintConsole()
{
	// Recurrence of the form T(n) = aT(n/b) + f(n)
	System.out.println("\nRecurrence Relationship");
	System.out.println("T(n) = " + str(a) + "T(n/" + str(b) + ") + " + fx.toString() );
}

public String toString()
{
	String aStr="";
	String bStr="";
	String fxStr="";

	// Recurrence of the form T(n) = aT(n/b) + f(n)
	if(fx.terms.size() == 0)
	{
		fxStr = "_";
	}
	else
	{
		fxStr = fx.toString();
	}

	if(a == -1.0)
	{
		aStr = "_";
	}
	else
	{
		aStr = str(a);
	}

	if(b == -1.0)
	{
		bStr = "_";
	}
	else
	{
```

```
                bStr = str(b);
        }



        return("T(n) = " + aStr + "T(n/" + bStr + ") + " + fxStr );
        //return("T(n) = " + str(a) + "T(n/" + str(b) + ") + " + fx.toString() );

}


public String toStyledString()
{
        String aStr="";
        String bStr="";
        String fxStr="";

        // Recurrence of the form T(n) = aT(n/b) + f(n)
        if(fx.terms.size() == 0)
        {
                fxStr = "_";
        }
        else
        {
                fxStr = fx.toStyledString();
        }

        if(a == -1.0)
        {
                aStr = "_";
        }
        else
        {
                aStr = str(a);
        }

        if(b == -1.0)
        {
                bStr = "_";
        }
        else
        {
                bStr = str(b);
        }



        return("T(n) = " + aStr + "T(n/" + bStr + ") + " + fxStr );
        //return("T(n) = " + str(a) + "T(n/" + str(b) + ") + " + fx.toString() );

}


public Double getA() {
        return a;
}
```

```
	public void setA(Double a) {
		this.a = a;
	}

	public Double getB() {
		return b;
	}

	public void setB(Double b) {
		this.b = b;
	}


	public String str(Double var)
	{
		if(Math.floor(var) == var)
		{
			Integer varInt = var.intValue();
			return varInt.toString();
		}

		return var.toString();
	}


}
```

File: Sign.java

```
package algorithmLibrary;
public class Sign
{
	boolean ispositive;
	public Sign(String p_sign)
	{
		if(p_sign.equals("+"))
			ispositive = true;
		else
			ispositive = false;
	}

}
```

File: TimeComplexity.java

```
package algorithmLibrary;
```

```
public class TimeComplexity {

	private ComplexityType complexityType;
	private FunctionX functionX;

	public TimeComplexity()
	{

	}
	public TimeComplexity(ComplexityType p_ComplexityType, FunctionX p_FunctionX)
	{
		complexityType = p_ComplexityType;
		functionX = p_FunctionX;
	}

	public String toString()
	{
		return complexityType.name()+"("+functionX.toString()+")";
	}

	public String toStyledString()
	{
		return complexityType.name()+"("+functionX.toString()+")";
	}

	public ComplexityType getComplexityType() {
		return complexityType;
	}
	public void setComplexityType(ComplexityType complexityType) {
		this.complexityType = complexityType;
	}
	public FunctionX getFunctionX() {
		return functionX;
	}
	public void setFunctionX(FunctionX functionX) {
		this.functionX = functionX;
	}

}
```

File: TimeComplexityMasterMethod.java

```
package algorithmLibrary;

public class TimeComplexityMasterMethod extends TimeComplexity {

	private CaseOfTheorem caseOfMasterTheroem;

	public TimeComplexityMasterMethod()
	{
		super();
	}
```

```
	public TimeComplexityMasterMethod(CaseOfTheorem p_casOfTheroem, ComplexityType
p_ComplexityType, FunctionX p_FunctionX)
	{
		super(p_ComplexityType, p_FunctionX);
		caseOfMasterTheroem = p_casOfTheroem;
	}

	public CaseOfTheorem getCaseOfMasterTheroem() {
		return caseOfMasterTheroem;
	}

	public void setCaseOfMasterTheroem(CaseOfTheorem caseOfMasterTheroem) {
		this.caseOfMasterTheroem = caseOfMasterTheroem;
	}

}
```

REFERENCES

[1]Introduction to Algorithms. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. The MIT Press. 2009.

[2]Algorithm Design. Jon Kleinberg, Eva Tardos. Pearson Education. 2005

[3]Taxonomy of Visual Algorithm Simulation Exercises, Ari Korhonen and Lauri Malmi. 2004.

[4]Exploring the Role of Visualization and Engagement in Computer Science Education, Thomas L. Naps, Rudolf Fleischer, Myles McNally. 2003.

[5]Design Pattern for Algorithm Animation and Simulation, Ari Korhonen, Lauri Malmi and Riku Saikkone. 2001.

[6]MatrixPro - A Tool for Demonstrating Data Structures and Algorithms ExTempore, Ville Karavirta, Ari Korhonen, Lauri Malmi and Kimmo St˚alnacke. 2004.

[7]Automatic Assessment of Exercises for Algorithms and Data Structures, Mikko-Jussi Laakso and Tapio Salakoski, 2004.