

# Time Complexity Estimator

A-Level Computer Science Project by Isaac Ling

## Contents

### *Analysis*

Background to Project .....	2
Research.....	2
Description of the Current System .....	3
Evidence of Analysis.....	4
Proposed Solution.....	5
Modelling of the problem .....	6
Objectives .....	9
System requirements.....	10
Acceptable Limitations.....	10
Changes.....	10
Appended Objectives.....	11

### *Documented Design*

Program Structure .....	12
High-Level Overview .....	13
Description of algorithms.....	18
Description of data structures .....	31
File design .....	34
Design of User Interface .....	35

### *Testing*

Testing.....	39
Evidence of Testing .....	44

### *Technical Solution*

Technical Solution .....	60
--------------------------	----

### *Evaluation*

Evaluation of objectives:.....	101
Feedback from my end-user .....	103

# Time complexity estimator

## Analysis

### Background to Project

In computer science theory, 'time complexity' is a critical piece of information when studying a particular algorithm and is the best indication of asymptotic behaviours of programs. The time complexity of an algorithm is defined as the relationship between the input of an algorithm, and the subsequent time said algorithm will take to complete. It is often represented in 'Big O' notation, a notation that describes the heuristic and generalised behaviour of an algorithm. The problem at hand is how to determine the time complexity of a given algorithm effectively, quickly, and reliably without manually looking through code. This is especially critical when dealing with code that has the implementation hidden, since there is no way to manually calculate the time complexity. The problem is relevant to almost all programmers, since the solution could help know where code is being bottlenecked, and how a program is going to withstand against relatively large inputs. It is important to mention that it is impossible to calculate the true time complexity of an algorithm 100% of the time, due to Rice's theorem, an extension of Turing's halting problem. It states that any non-trivial (that is, not deterministically true or false) property of a language which is recognised by a Turing machine is undecidable<sup>1</sup>, time complexity being such a property. This explains the lack of systems in place to get a reliable solution to the outlined problem. However, estimating the time complexity of relatively small algorithms is possible and should be reasonably reliable. Therefore, the aim of this project is to write a program which will take an algorithm as an input and estimate its time complexity.

### Research

After researching the subject some more, I have made the following observations:

'Complexity' of an algorithm refers to resource usage of an algorithm, and how this changes relative to the input size. Time complexity is a specific complexity that describes the relationship between the input size into an algorithm and the time it takes to compute this. We denote time complexity as a function of input size, with time being the dependant variable. The most common way to write this is using Big O notation, where O is said to be a function of input size. Big O notation differs slightly, however, as Big O is strictly the asymptotic behaviour of the algorithm, that being the limit of time complexity when the input is large. This is extremely useful as this gives an indication of how the algorithm will react to large inputs, compared to its performance with small inputs. It is critical to remember that time complexity gives no indication to the runtime of the algorithm. A constant time algorithm is said to be the most efficient time complexity, yet the algorithm could still take a thousand years to run. This is why time complexity is not a measure of performance, merely a measure of the *change* in performance with respect to input.

Big O notation is not the only measure of time complexity. There also exists two others, big theta and big omega. An overview of each is given below:

Big O is the measure of the worst-case time complexity of an algorithm. This is generally the most used as arguably an algorithm is only as good as its worst-case scenario, as an efficient Big O guarantees that the algorithm will always be at least this efficient.

Big Theta (Big  $\Theta$ ) is the measure of the most accurate average time complexity of an algorithm. This is very complicated to calculate, and is arguably not as useful, since you cannot know how big the range of possible time complexities it covers. If you were designing a website and wanted to know how an algorithm will cope with a large number of users on the website at once, Big  $\Theta$  is of little importance,

# Time complexity estimator

## Analysis

as you want to ensure that not just the majority of users will get a good experience from the website, but all users do.

Big Omega ( $\Omega$ ) is the measure of the best case for time complexity of an algorithm. Together with Big O notation, you get the range of possible asymptotic behaviours of the algorithm, and you can therefore see how an algorithm will act at large inputs in all cases.

I have chosen to use Big O notation in my project as this is the most commonly used, and therefore most understood and recognisable. It is also in my opinion the most useful measure of time complexity.

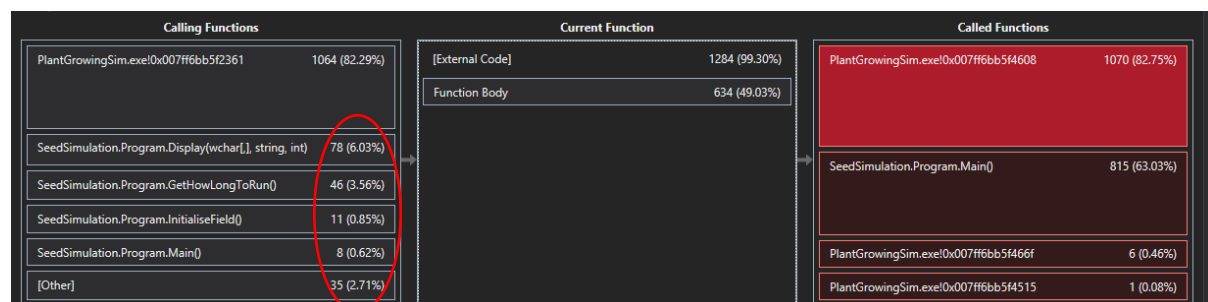
## Description of the Current System

The systems that get closest to the problem above are 'code profilers', pieces of software that will run your code and feedback with several pieces of important data, such as where the program runs slowest and how much memory the program is using. These are helpful for individual runs of an algorithm, but it is difficult to see how this data changes with different inputs without manually running it several times and recording the data yourself. None that I have used have ever shown an estimated time complexity, probably because profilers aim to give an overview of the entire code, and not detailed feedback on individual algorithms.

The code profiler which I personally use is the Visual Studio 2019 Performance Profiler, developed by Microsoft. It is part of the debugging options it provides and offers multiple different areas of your program to get data on.

The most relevant is the CPU usage profiler option, as it shows where time is spent within your program and dividing this by the number of times it was called gives you an estimate of how long each call to your function is roughly taking. If you run your program with differing inputs and comparing the times each call to the function takes, you can get an idea of how the time changes with respect to the input size. This is a laborious process and is unreliable. This is not necessarily the fault of the profiler, since it is not designed to estimate time complexity, but it does mean that there is not an easy way to scale up functions built into Visual Studio.

The screenshot below shows the information given about time for different functions, specifically the percentages circled show the percent of the overall runtime that each function used. It is also possible to use the 'Stopwatch' built in class for C#, and manually measure the time it takes to complete certain parts of your program. This is by no means a quick process, and still requires you to run the program several times with different inputs to understand how that code will react to large inputs.



To sum up, the current system is wholly inadequate for measuring time complexity, so it is clear that there is a need for a practical solution.

# Time complexity estimator

## Analysis

### Evidence of Analysis

I have identified the end users as hobbyist or professional programmers, as optimising and profiling code is essential in all programs. With this in mind, I have found a specific end user as a further maths teacher who writes plenty of code.

I conducted the following interview with said further maths teacher to gather valuable information to justify my objectives. Since there is no mainstream solution to this problem, I have asked questions not on problems with the current solution, but on what a successful solution might look like. After each answer I have written a brief summary of how I plan to implement this into my final program:

Q1: Do you believe that time complexity is a valuable piece of data when analysing a program, and why?

A1: *"Yes, although more important for some than others. For some programs you need to have an idea of how long they will take to run once you scale up the problem you are trying to solve. If it is going to take years to run, you might decide it's not worth it."*

It is evident that this program has a practical use, and the teacher pointed out the most obvious one, scaling up times for a given program.

*[These questions are in relation to a program which could find the time-complexity of an algorithm, along with some performance data]*

Q2: How would you want the time complexity and performance data to be outputted to you so that you could clearly understand the result?

A2: *"An estimate,  $k$ , of the order  $O(n^k)$  and some example times for  $n = 10, 100, 1000$  in sensible units, days, hours, mins, seconds, milliseconds etc"*

This I shall achieve by plotting the results on a graph and then evaluating a line of best fit. This will give 'example times' for all the values in between all the inputs the program already measured to estimate the time complexity. In terms of sensible units, I will ensure I scale both axis enough that all points will fit on nicely. The points that the program used to estimate the time complexity can be interrogated to view the input and subsequent time.

Q3: How should this output be customisable by the user?

A3: *"It would be useful to select a number of  $n$ 's and/or a drop down list, e.g.:*

*2, 4, 8, ...*

*10, 100, 1000, ...*

*with a choice of how many, e.g., 6 would result in 2, 4, 8, 16, 32, 64"*

To achieve this graphically, I will allow the user to scale the axis to any value so that they can specifically focus in on some specific values of  $n$ .

Q4: Which time complexities would be the most important for this program to identify?

A4: *Of the order  $O(n^k)$*

This time complexity is also called 'Polynomial', and with varying values of  $k$  it can include constant, linear, quadratic, and cubic. I shall also consider logarithmic and perhaps exponential should I have time.

# Time complexity estimator

## Analysis

Q5: How should this program make it easier to compare different algorithm's performances?

A5: *"An estimate of the order for each algorithm and example comparative times for same n's"*

This answer can be integrated into my graphical solution by having the ability to plot multiple algorithm's results on the same axis.

Q6: What extra tools would be desirable for you to be able to gain useful information from the output?

A6: *"After you have an estimate of the order, allow the user to input a value of n to estimate the time the algorithm might take"*

I want extrapolation of the results to be included in my project, as it adds a practical element to the program that allows you to scale up the results, which also solves the problem my end-user identifies in A1.

Q7: What would be the maximum amount of time you would be willing to wait for the program to calculate the time complexity before you got the results?

A7: *"I would be okay with overnight, so say 24 hours."*

This is a little longer than I would like, so I'm going to aim for around 30 seconds. This should be sufficient time to get the accuracy required in A8.

Q8: What would be the minimum acceptable rate for the program to correctly identify the time complexity?

A8: *"80%"*

I would like this to be met by all small and simple algorithms for all time complexities.

## Reflection on interview

It is clear from the interview that being able to compare different algorithms is critical. Whilst I have deviated slightly from the suggested solution, I believe my method meets all of the criteria and adds a visual element in the form of a graph. From these answers, it is clear that this problem is also relevant. I believe that extrapolation is also a key feature, as it seems to be the most useful piece of information that can be found from the time complexity. I have used these answers to design a solution to the problem which is outlined below.

## Proposed Solution

I have devised a solution that meets all the requirements set out by my end-user. The abstracted solution goes as follows:

1. A main menu allows the user to enter a filename for a C# file
2. This file will be compiled to an executable
3. This executable will be run with varying inputs, and the time taken to compute these will be recorded
4. Using these times, the time complexity will be estimated
5. The time complexity will be printed out along with a graph of the times taken for each input
6. The user can run another C# file and it will be drawn on the same graph as the previous file

# Time complexity estimator

## *Analysis*

### Modelling of the problem

With the proposed solution as listed above, I have created some pseudocode that will act as a reference for my technical solution. It does not go in depth on how I will calculate the time complexity, as this is described afterwards.

```
results ← []
```

```
FUNCTION Welcome()  
    file ← ""  
    PRINT("Welcome to the Time Complexity Estimator")  
    PRINT("Enter the program:")  
    file ← INPUT()  
    CompileFunction(file)  
ENDFUNCTION
```

```
FUNCTION CompileFile(file)  
    algorithm ← ""  
    algorithm ← file.ConvertToEXE()  
    RunAlgorithm(algorithm)  
ENDFUNCTION
```

```
FUNCTION RunAlgorithm(algorithm)  
    times ← []  
    input ← 1  
    FOR i ← 0 TO 10 DO  
        input ← 100 * i  
        Timer.Start()  
        algorithm.Run(input)  
        Timer.End()  
        times[i] ← Timer.Time  
        Timer.Reset()  
    ENDFOR
```

```
        CalculateTimeComplexity(times)  
ENDFUNCTION
```

```
FUNCTION CalculateTimeComplexity(times)  
    result ← ""  
    result ← times.CalculateTimeComplexity()  
    results.Add(result)  
    PrintResults()  
ENDFUNCTION
```

# Time complexity estimator

## *Analysis*

```
FUNCTION PrintResults()  
    addResult ← false  
  
    FOREACH result IN results DO  
        PRINT("The time complexity is " + result)  
    ENDFOREACH  
  
    PRINT(results.ToGraph())  
    PRINT("Add a result?")  
    addResult ← INPUT()  
  
    IF (addResult) THEN  
        Welcome()  
    ELSE  
        QUIT()  
    ENDIF  
ENDFUNCTION
```

## **Welcome()**

### *Potential methods of calculating time complexity*

To calculate the time complexity, irrespective of the method, sufficient data needs to be gathered on the algorithm. This data, at the most fundamental level, will need to include a list of inputs along with a list of times it took to calculate with these inputs. Using this data, I have identified these possible methods of estimating time complexity, along with some positives and negatives:

#### Neural Network

If the inputs and times were plotted on a graph, a neural network could be implemented that could estimate the equation of the curve created. Finding the dominant term in this equation would yield the Big O for that algorithm

#### *Positives:*

- Only one implementation necessary, that is, this method would work for all time complexities
- Once the neural network has been calibrated, it would be relatively quick to identify time complexity
- Has potential to be very accurate

#### *Negatives*

- Very difficult to implement
- Only slight deviations from the correct graph could cause it to identify the time complexity incorrectly

# Time complexity estimator

## *Analysis*

### Linearisation

Linearisation is the process by which a function is manipulated to take the shape of a linear function by changing the axis. This could be achieved by taking the data from the inputs or times and applying a function to each one and then checking if the function created is linear.

#### *Positives*

- Reasonably easy to implement
- Works on almost all functions

#### *Negatives*

- Still requires an algorithm to determine if it becomes linear
- Slow to run, since it would need to check against every possible function to see if it becomes linear

### Calculus

When differentiating a polynomial function, its order decreases. There could be a way of repeatedly differentiating a function until it becomes constant to determine its order.

#### *Positives*

- Should work with all polynomial functions
- Has the potential to be very accurate

#### *Negatives*

- Does not work with non-polynomial function
- Still requires an algorithm to determine if it becomes constant

### Statistical methods

Finding how well the times 'fit' with a known function's data using statistical methods could identify which function is the closest to the data

#### *Positives*

- Could theoretically work on any function, so no other algorithms are required

#### *Negatives*

- Generally, this is quite inaccurate unless the volumes of data are huge, which is impractical
- Slow to run, as it would need to find its 'fit' for all possible time complexities to determine which one is the closest.

I have chosen to use a combination of calculus, linearisation and statistical methods. Calculus I shall use for polynomials, linearisation for logarithmic and statistical methods for determining when a function becomes constant. I have chosen to not use a neural network as I believe that the extra complications in setting it up provide no added benefits to the methods described above.



# Time complexity estimator

## Analysis

### Objectives

1. A Main Menu appears on start-up
  - 1.1. Option to display information about program
    - 1.1.1. The information provides instructions on how to use the program
    - 1.1.2. The information outlines the purpose of the program
  - 1.2. Option to input an algorithm
  - 1.3. Option to quit the program
    - 1.3.1. Asks the user to re-affirm their selection before the program quits
      - 1.3.1.1. Should the user request to go back, the program will return to the Main Menu
2. User can input algorithms
  - 2.1. Inputted as a .cs file
    - 2.1.1. Will recognise if the user has entered a file extension
      - 2.1.1.1. Will find the file irrespective of whether a file extension is added or not
  - 2.2. Will check if the file exists
    - 2.2.1. A warning will be shown if the file does not exist
  - 2.3. Allows the user to return to the Main Menu when 'quit' is entered
3. Build and run the algorithm
  - 3.1. Build the .cs file into an .exe file and place it in the same location
  - 3.2. Display an informative progress bar
    - 3.2.1. Will show percentage next to progress bar
  - 3.3. Will handle errors within the user's algorithm and display them
    - 3.3.1. Displays the algorithm's error codes within the program
4. Calculates time complexity of algorithm
  - 4.1. Dynamically calibrate input sizes to ensure the program is as quick as possible
  - 4.2. Can differentiate between at least five time complexities
    - 4.2.1. Can determine constant time
    - 4.2.2. Can determine logarithmic time
    - 4.2.3. Can determine linear time
    - 4.2.4. Can determine quadratic time
    - 4.2.5. Can determine cubic time
  - 4.3. Is at least 80% accurate in all time complexities for small algorithms
5. Display results in a graph
  - 5.1. Draw a graph to the screen
    - 5.1.1. Draws the axes
      - 5.1.1.1. Both axes labelled
        - 5.1.1.1.1. Input axis is scaled so that it is easier to read
      - 5.1.1.2. Axes can be scaled by the user
      - 5.1.1.3. Axes can be re-sized by the user
    - 5.1.2. Plots the 'nodes' from the algorithm's result data
    - 5.1.3. Plots a 'curve of best fit' for the algorithm's 'nodes'
      - 5.1.3.1. The curve is calculated using equations that are generated from the algorithms time complexity
  - 5.2. Graph can be extrapolated or interpolated by the user
    - 5.2.1. Can extrapolate time
    - 5.2.2. Can extrapolate input

# Time complexity estimator

## Analysis

- 5.2.3. Generates result from solving the equation from 5.1.3.1
- 5.3. Graph can be interrogated by the user
  - 5.3.1. The user can move through the 'nodes'
    - 5.3.1.1. Information will be printed about the current node
- 5.4. Option to add an algorithm onto the same graph
  - 5.4.1. Up to 5 plots on one graph
    - 5.4.1.1. Prints out a suitable message should the user try to add more
  - 5.4.2. Plots are colour-coordinated to make the graph more readable
  - 5.4.3. All graph options can be applied to all algorithms
- 6. All inputs are suitably handled and validated
  - 6.1. The program should handle all invalid inputs by printing a suitable message
    - 6.1.1. The program will continue after an invalid input is entered
- 7. Program will check for prerequisites
  - 7.1. Will return a suitable message should they not be installed

## System requirements

Through some research on how a C# program could compile and run another '.cs' program, I have identified some system requirements:

1. The Microsoft Windows operating system must be used, this is due to the use of the command prompt, the Windows shell. This could be modified to include Linux by accessing the Linux shell, but this is beyond the scope and purpose of the program
2. .NET Framework needs to be installed, as this is how the program will compile the '.cs' to an '.exe'

## Acceptable Limitations

As already mentioned, Rice's theorem proves that a perfect solution to this problem is impossible, so it cannot be expected that this program will be correct 100% of the time. It has been agreed with my end user that an 80% success rate with small algorithms is acceptable.

Another limitation is with operating systems and is outlined in the 'System Requirements' section. Since this program is heavily reliant on the Windows command prompt, it is not possible to run this on any other operating system. I believe that this is not a harsh limitation since, at the time of writing, around 73% of desktop PCs are using the Windows OS<sup>i</sup>.

The final limitation is with the amount of time complexities the program will be able to calculate. Polynomial functions with an order higher than 3 are very similar to a cubic until very large inputs. This would take a long time to compute, so I believe it is justified to only support time complexities up to cubic. Exponential functions, which were added later, will cover all functions above cubic.

## Changes

After coding some of the solution, the following changes were made to my original objectives:

1. Due to having spare time at the end, I have included exponential time complexity support. I plan for this complexity to also run with the same 80% success rate as the other complexities.
2. An extra feature was added at the end, namely the ability to find the intersection between two graphs. The extra objectives this creates are listed below.

# Time complexity estimator

## *Analysis*

### Appended Objectives

8. Intersections between graphs can be calculated
  - 8.1. User picks two graphs to find the intersection between
    - 8.1.1. Should there not be enough graphs to find an intersection between, a suitable error message is printed
    - 8.1.2. The user cannot pick the same graph twice
  - 8.2. Gives the input and time of the intersection
    - 8.2.1. If there is no intersection, a suitable error message is printed
9. Can determine exponential time

---

<sup>i</sup> <http://kilby.stanford.edu/~rvg/154/handouts/Rice.html>

<sup>ii</sup> <https://www.statista.com/statistics/218089/global-market-share-of-windows-7/>

# Time complexity estimator

## *Documented Design*

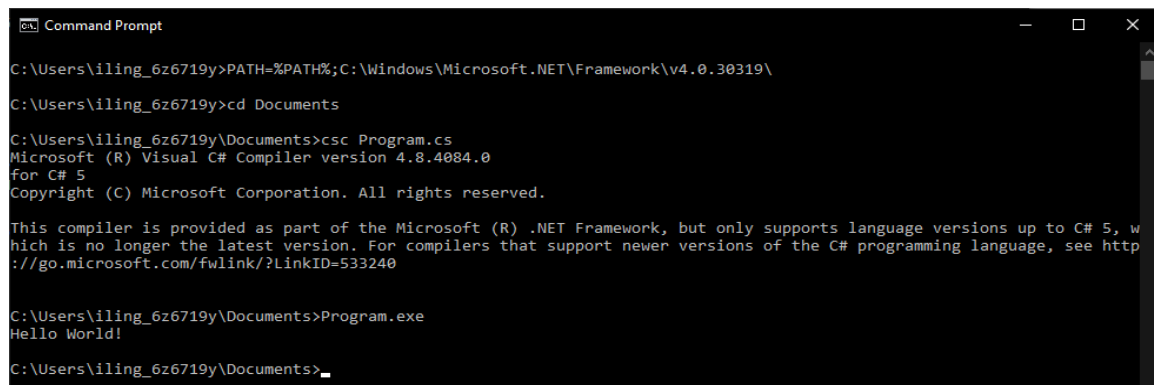
### Program Structure

Through the modelling in my analysis section, specifically the pseudocode, I have identified 4 main parts of the program:

1. Compiling and running the C# file
2. Calculating its time complexity
3. Printing the results
4. Interrogating the results

Point 2 has been investigated as part of the modelling in the analysis section, but the others will be explained here:

1. Through some research, I have found that it is possible to access the Command Prompt from within C# code. This is helpful as it is possible to compile a C# program to an exe through the 'csc' or C# Command Line Compiler command. By accessing the command line and entering the command 'csc [Filename].cs' the file should be converted into an exe and placed back into the same directory. To perform this command, I will need to set a path to the Visual Studio folder as this contains the 'csc' executable. As a test, I compiled a test C# script 'Program.cs' to an .exe and then run it as seen here:



```
Command Prompt
C:\Users\iling_6z6719y>PATH=%PATH%;C:\Windows\Microsoft.NET\Framework\v4.0.30319\
C:\Users\iling_6z6719y>cd Documents
C:\Users\iling_6z6719y\Documents>csc Program.cs
Microsoft (R) Visual C# Compiler version 4.8.4084.0
for C# 5
Copyright (C) Microsoft Corporation. All rights reserved.

This compiler is provided as part of the Microsoft (R) .NET Framework, but only supports language versions up to C# 5, which is no longer the latest version. For compilers that support newer versions of the C# programming language, see http://go.microsoft.com/fwlink/?LinkID=533240

C:\Users\iling_6z6719y\Documents>Program.exe
Hello World!
C:\Users\iling_6z6719y\Documents>
```

As you can see, these commands are enough to build and then run the program.

2. Calculating the time complexity is described in the Analysis section.
3. I shall print the time complexity in Big O notation alongside the expanded English version for users who perhaps are not familiar with Big O. As mentioned in my analysis section, I believe a graph is the best way to meet my end-user's requirements. This shall be printed underneath the time complexity information.

When thinking about how I would show the results to the user, I researched the possible applications I could use to run my program on. Windows Forms provides a GUI (Graphical User Interface) environment whilst the C# console provides a CLI (Command Line Interface). I decided against the GUI as I believe that it would take longer to learn and use, and I would prefer to use this time to add more functionality to my program. The graph will have to be drawn using Unicode characters, but this should not be a problem.

4. As agreed with my end-user, extrapolation is a key part to this program. This shall be implemented such that both inputs and times can be extrapolated. This means that a user could either find how long the program will take for a particular input, or they could find the input that would be needed for the algorithm to take a specific amount of time. Another option that I would like to be implemented is the ability to look at each point on the graph and see its time

# Time complexity estimator

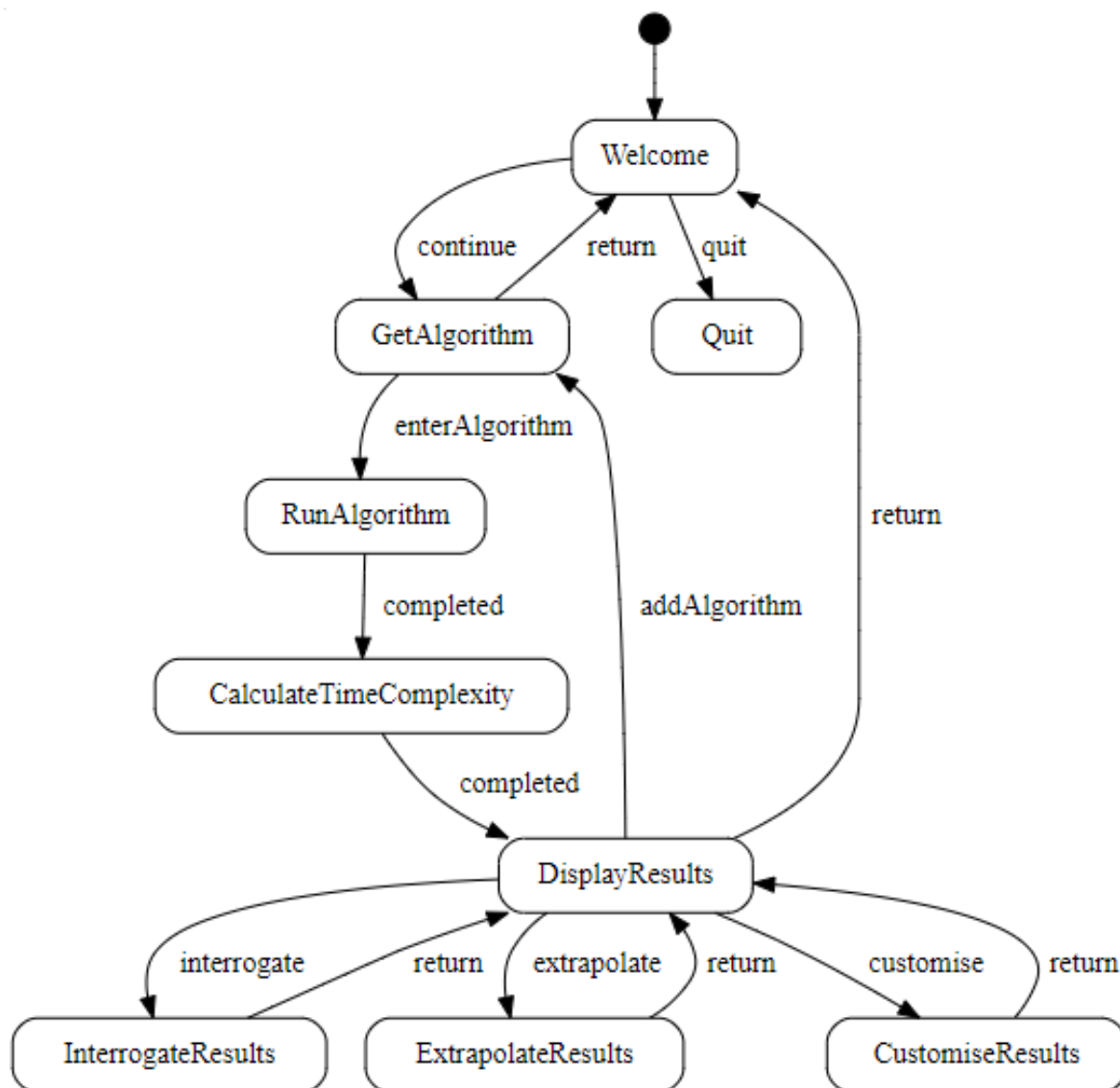
## Documented Design

and input. A necessary utility feature would be the ability to rescale and resize the axis of the graph, so that people with poorer vision are still able to customise the graph to their liking. The rescaling of the axis means that it can be zoomed out or in, allowing the user to see the long term trends of the program.

## High-Level Overview

### State transition diagram

This state transition diagram shows the overall flow of the program, and how each state can be accessed from other states. This is the most abstracted overview of how the program works as a whole and can be used as a reference when navigating other high-level overviews.



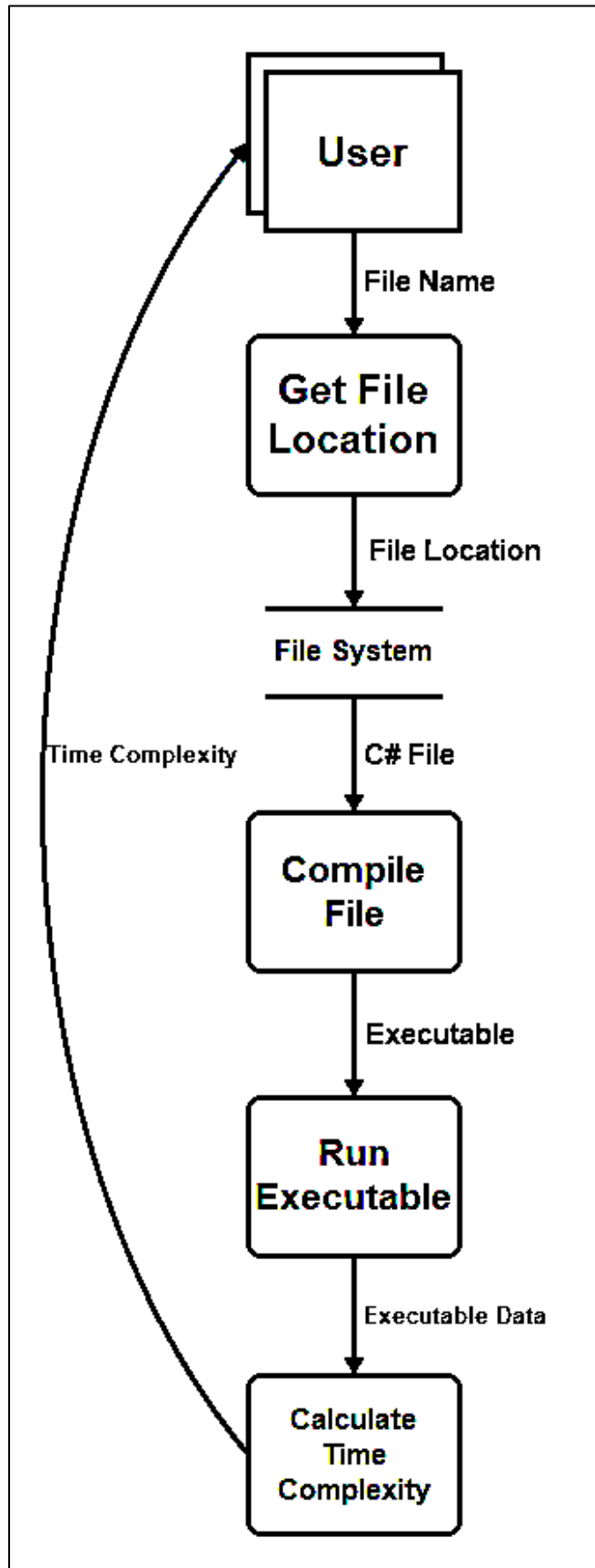
Please note that there have been several changes to this diagram:

1. The addition of several new graph options
2. The addition of an information page

# Time complexity estimator

## Documented Design

### Data Flow Diagram

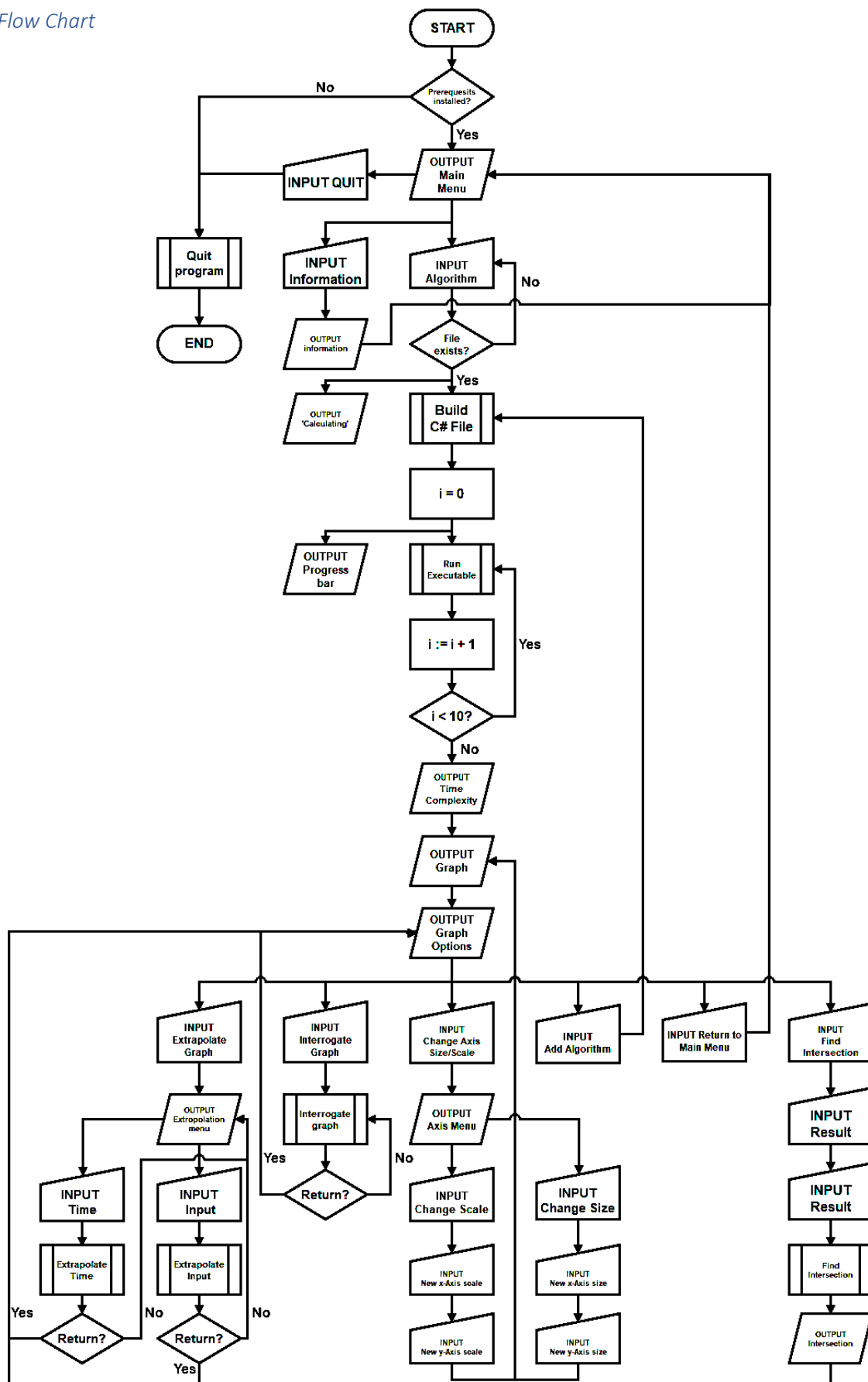


This data flow diagram makes it very clear that the purpose of this program is to turn a file into some useful information that is then relayed back to the user. The only data store is the user's file system, which is local to the user's computer. This means that any form of internet connection is not necessary.

# Time complexity estimator

## Documented Design

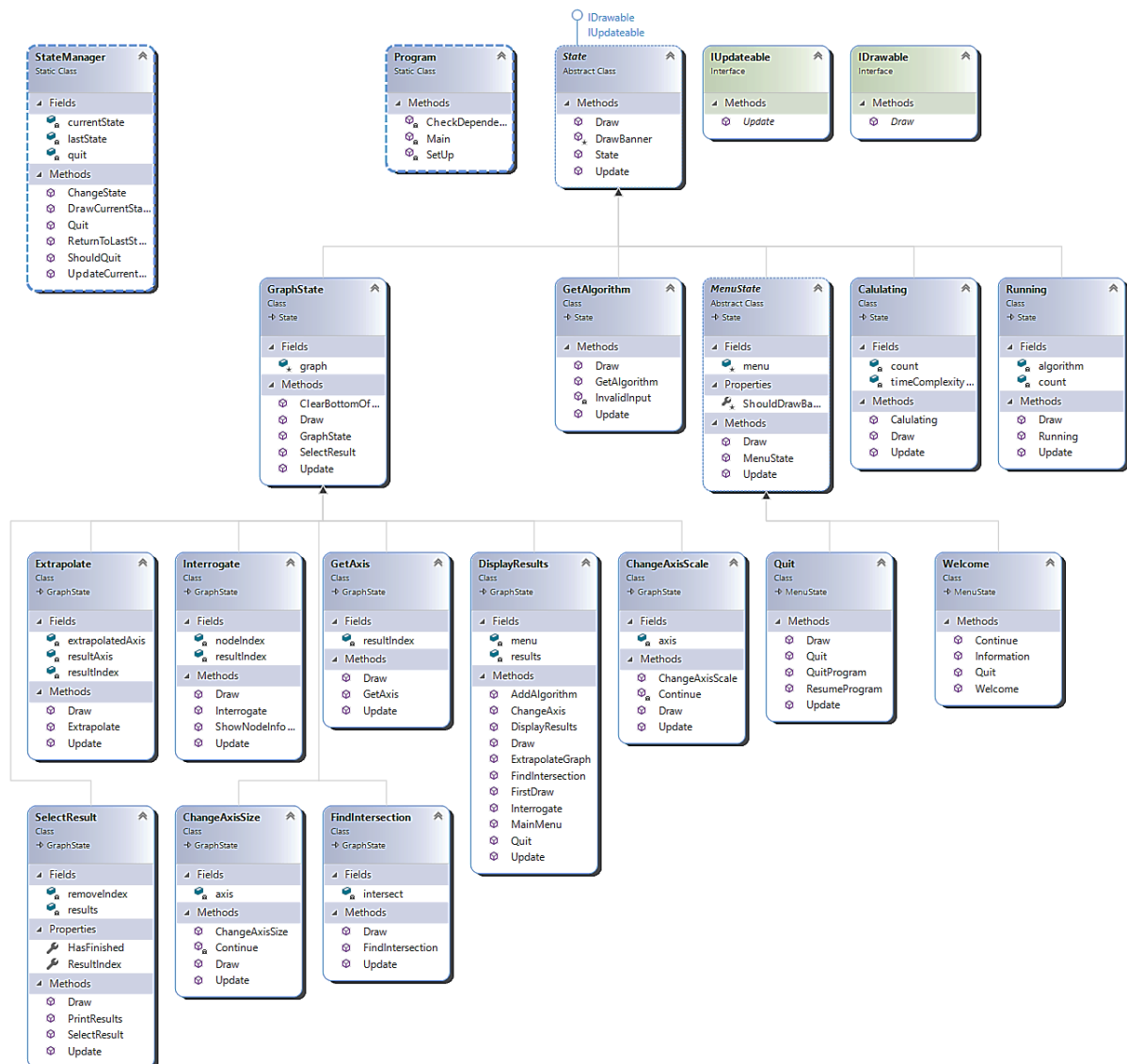
### Flow Chart



# Time complexity estimator

## Documented Design

### Class diagrams

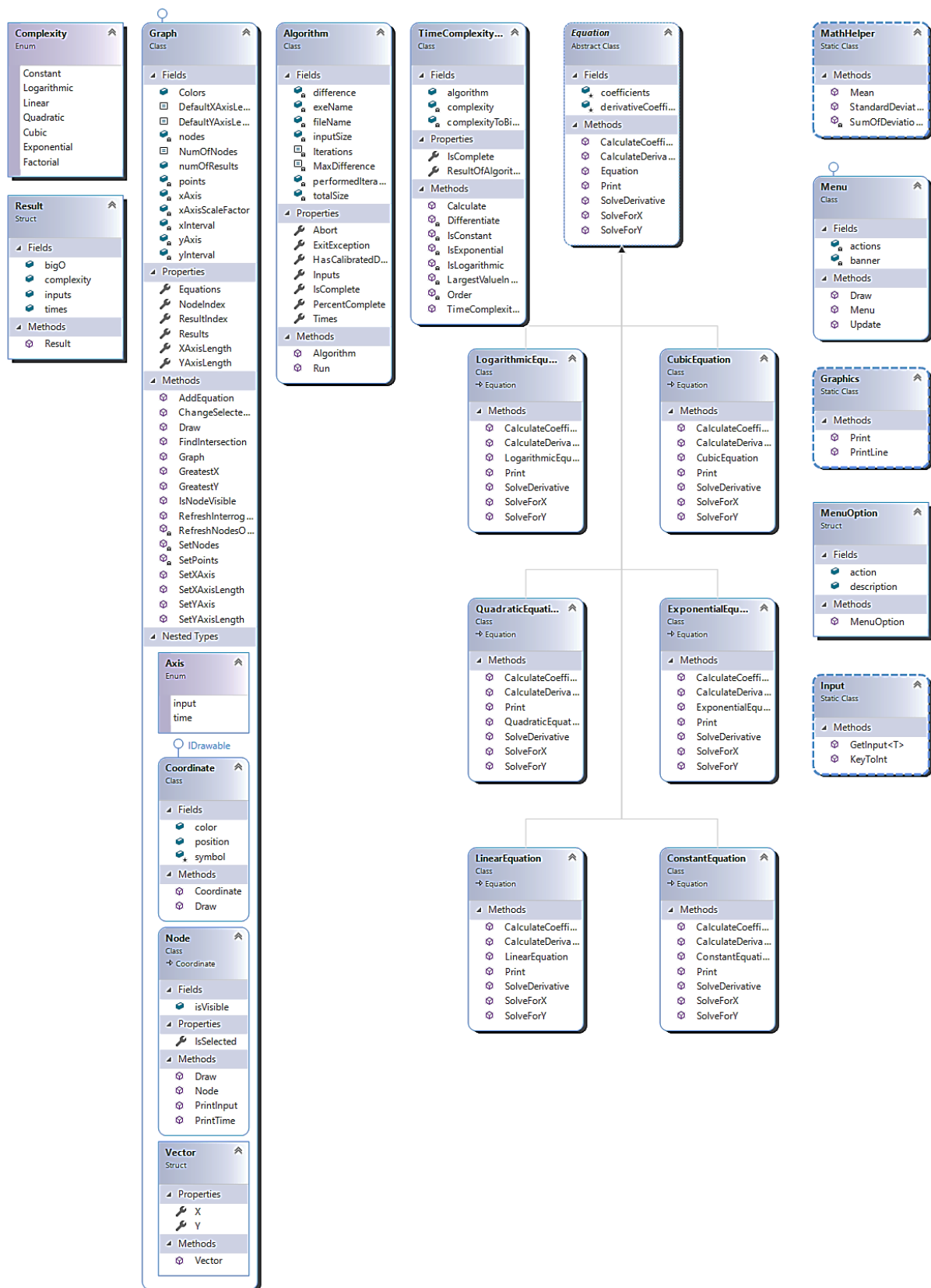


I chose to use a 'state' system to separate out the distinct sections of my program. This makes it highly maintainable, loosely coupled and modular, so extra parts can be added on with ease. They all inherit from an abstract State class that implements both the IDrawable and IUpdateable interfaces. Many states require a graph to be used and manipulated, so there is a 'GraphState' that is an abstract class that contains helper methods and a graph field. This is similar to the 'MenuState', which is for states that use a menu to choose an option from. All states can be moved between with the aid of the 'StateManager' static class. The 'StateManager' class also provides a way to exit the program via the 'Quit' method. It also holds the last state that the program was in to easily move back to that state. Separating out draws and updates makes it easier to add new code and allows each part to be used independently, for example, you could redraw a state without having to update it. This would not be possible if I had a block of code that both printed and updated it in one. The graph class contains four nested types, namely coordinate, node, vector and axis. These are nested as they are never to be used outside the context of a graph. This form of encapsulation ensures that the separate classes are as loosely coupled as possible, as restricting the scope of them means that they are hidden from other classes.



# Time complexity estimator

## Documented Design



# Time complexity estimator

## *Documented Design*

### Description of algorithms

#### *Algorithm to build and get data on an inputted program*

To be able to estimate the time complexity of a program, we have to gather some relevant data on the runtime of the program. Since time complexity is the asymptotic behaviour of the program, we will need to run the program multiple times with different inputs and compare the time it takes to compute each of them. The devised solution to this is to run the program 10 times with different inputs. The inputs will need to be calibrated so that more complex algorithms will not take a large amount of time to complete. The calibration is aiming to find the largest input that can be plugged into the program before it takes over 7.5 seconds. The reason for wanting the largest input is to have the largest spread of inputs as possible, so that the behaviour of the program can be most accurately predicted. The pseudocode for calibrating inputs is:

```
inTime ← true
difference ← 1
iterations ← 10
maxDifference ← 1000000

WHILE (inTime && difference <> maxDifference)
    timer.Start()
    algorithm.Run(difference * iterations)
    timer.End()

    IF (timer.elapsedTime < 7500) THEN
        inTime ← true
        difference := difference * 2
    ELSE
        inTime ← false
    ENDIF
ENDWHILE
```

This will calibrate 'difference' such that the final input entered to the program (i.e., 10 \* difference) will take less than 7.5s. It can be safely assumed that all inputs below this will also be below this time threshold since algorithmic time complexity is always considered to be an increasing function.

Throughout this document, the term 'inputs' means the set of values that were inputted into the program, 'times' refers to the subsequent set of times that the algorithm took to compute the set of inputs, and 'nodes' refer to the coordinates that have an x component of an input and a y component as the time it took to calculate that input.

# Time complexity estimator

## Documented Design

### Algorithm to find the order of a polynomial

The majority of time complexities can be expressed in the form  $ax^n + bx^{(n-1)}$  etc, for example constant is when  $n = 0$ , linear is when  $n = 1$  etc. This was also the time complexity my end-user stated as being the most important for my program to be able to identify. The algorithm which I thought of to find the value of  $n$  for a given list of times with their respective inputs was repeated differentiation of the function until it is reduced to a constant. The general equation for a polynomial I define as:

$$f(x) = \sum_{r=0}^{n+1} a_r x^{n-r}$$

The derivative of this polynomial, using the power rule, I calculated was:

$$f'(x) = \sum_{r=0}^n a_r (n - r) x^{n-r-1}$$

From this, I derived the  $k$ 'th derivative of the polynomial:

$$f^{(k)}(x) = \sum_{r=0}^{n+1-k} \left( a_r \prod_{s=0}^r (n - r - s) \right) x^{n-r-k}$$

It is important to note that if  $n = k$ , then  $f^{(k)}(x)$  becomes a constant:

$$n = k \Leftrightarrow f^{(k)}(x) = \sum_{r=0}^1 a_r \cdot n! \cdot x^{n-r-k} \Leftrightarrow f^{(k)}(x) = a_0 \cdot n!$$

Therefore, for any polynomial of type  $f(x)$ , the order of the polynomial,  $n$ , is equal to the number of derivatives it takes,  $k$ , to reduce the function to a constant. This means that if a program could differentiate the times that an algorithm takes to complete with a constant increase in input, it could find the order of the time complexity by checking when it becomes constant. This can be demonstrated in the recursive pseudocode:

```
FUNCTION Order(times, depth)

    derivativeOfTimes ← Differentiate(times)

    depth := depth + 1

    IF (derivativeOfTimes is constant) THEN
        RETURN depth
    ELSE
        RETURN Order(derivativeOfTimes, depth)
    ENDIF

ENDFUNCTION
```

# Time complexity estimator

## *Documented Design*

This algorithm will return the integer value of 'n', or the order of the polynomial. This algorithm will only output the highest exponent, which is important as this is how time complexity is defined. From this value of n, the Big O will be  $O(x^n)$ .

### *Algorithm to differentiate a function*

In the pseudocode above, the implementation of differentiating a function is not described. The method to do this is called 'Discrete Differentiation'. Traditional differentiation is considered continuous as it covers all values of x, but since we only have a finite amount of data, another method is required. Recall that:

$$\frac{dy}{dx} = \lim\left(\frac{\Delta y}{\Delta x}\right)$$

Removing the limit, since we are not calculating the continuous derivative gives

$$\frac{\Delta y}{\Delta x}$$

This is defined as an increment of y divided by an increment of x. Therefore, for a discrete collection of times with a fixed difference in input between them, the discrete derivative is a list of the differences in times. Dividing by the change in x is not required if the differences between each item is constant, as it would simply scale each item up or down by the same amount. Here is an example collection of times, all of which have been generated from a fixed increase of input:

{1, 4, 9, 16, 25, 36}

Differentiating this produces a new collection of data that will be one element shorter. Each element is the difference between two adjacent times in the original collection:

{3, 5, 7, 9, 11}

This is the discrete derivative of the times. An implementation of this can be seen below:

```
FUNCTION DiscreteDerivative(times)
    derivative ← []

    FOR i ← 0 TO LENGTH(times) - 1 DO
        derivative[i] = times[i + 1] - times[i]
    ENDFOR
ENDFUNCTION
```

# Time complexity estimator

## Documented Design

### Algorithm to determine if a function is logarithmic

This can be achieved through the process of 'linearisation', a method of reducing a non-linear function into a linear one by manipulating one of the axis. For a logarithmic function, using an exponential y axis or a logarithmic x axis will form a straight line:

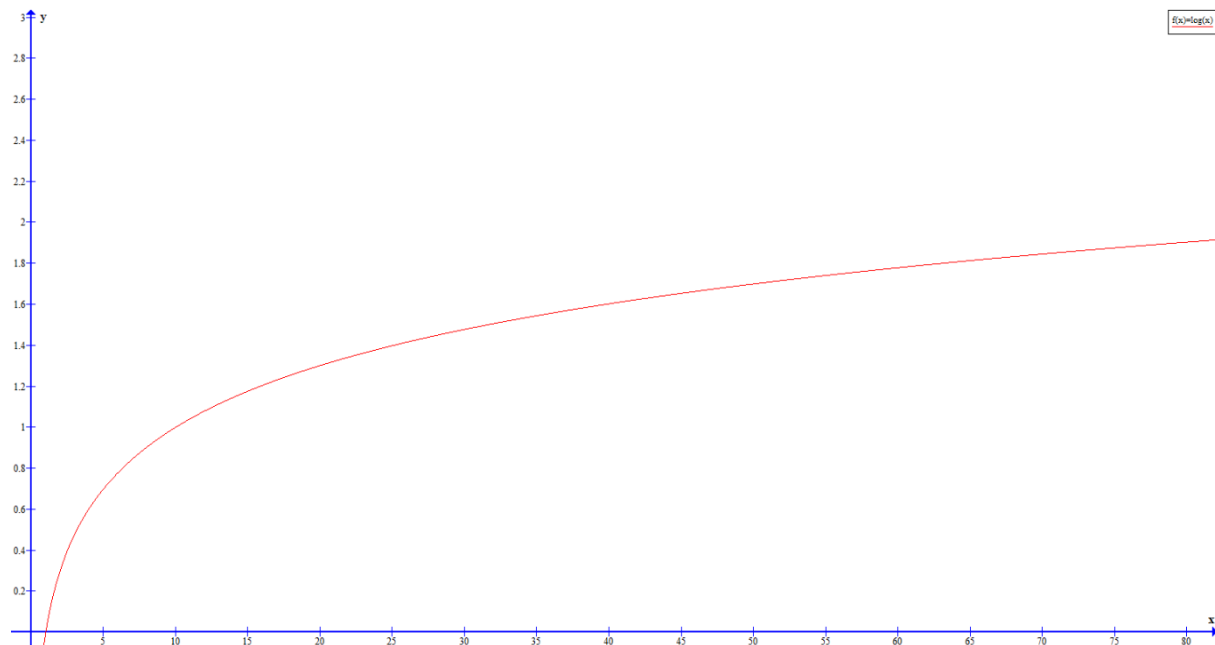


Figure 1: Logarithmic graph with standard axes

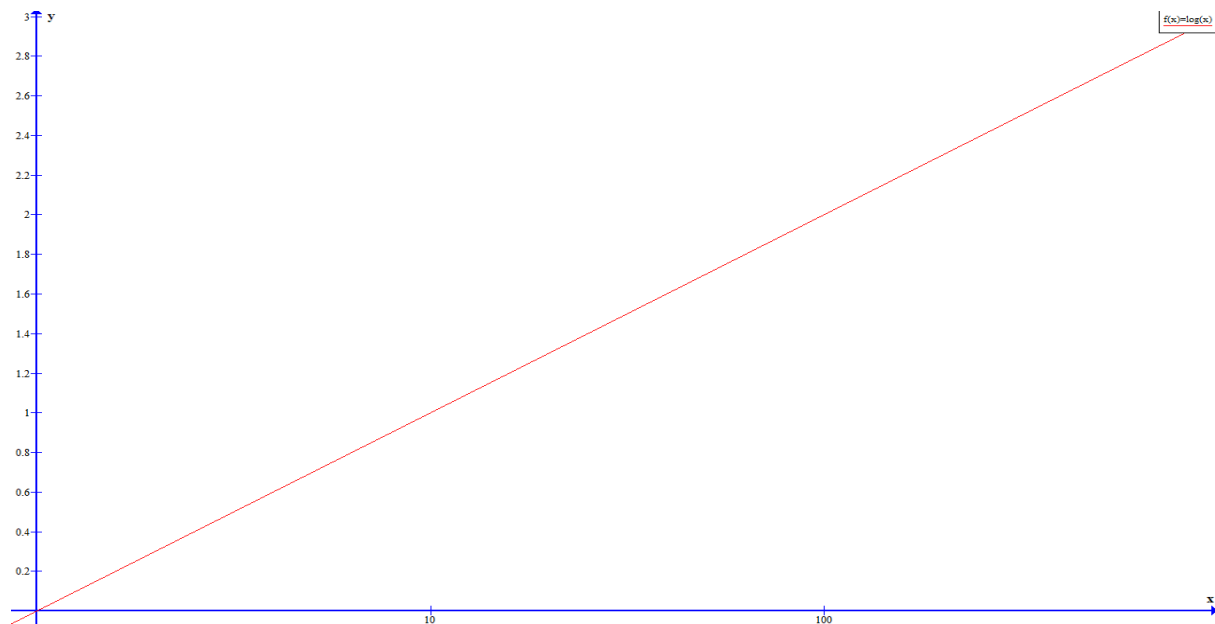


Figure 2: Logarithmic graph with a logarithmic x axis

To achieve this algorithmically, a constant would be raised to each of the times generated from running an algorithm, then these numbers would be passed into the polynomial algorithm specified above. Should this return 1 (i.e., linear), then it is likely that the algorithm was logarithmic. This is shown in the pseudocode:

# Time complexity estimator

## Documented Design

```
FUNCTION IsLogarithmic(times)

    FOR i ← 0 TO LENGTH(times) DO
        alteredTimes[i] ← 2 ^ times[i]
    ENDFOR

    IF (Order(alteredTimes) == 1) THEN
        RETURN TRUE
    ELSE
        RETURN FALSE
    ENDIF

ENDFUNCTION
```

### Algorithm to determine if a function is constant

The pseudocode for determining the order of a polynomial contains the line 'IF (derivativeOfTimes is constant)', which can be achieved by using the statistical value 'Standard Deviation'. The standard deviation of a set of numbers is defined as the square root of the sum of deviations (i.e., the distance each number is from the mean) squared. In a compact formula this is:

$$\sigma = \sqrt{\frac{\sum (x_i - \mu)^2}{N}}$$

The standard deviation of a set of constant numbers is evidently 0, so an algorithm that could calculate the standard deviation of the times generated by my program would be able to tell if it was constant by whether it was zero. Of course, minor fluctuations in times will occur due to CPU temperature changes, background tasks etc, so a technique to find the relative standard deviation is required. The method I have devised is to check the standard deviation at each derivative and compare it to the last standard deviation. If it has decreased, we have gotten closer to reducing the function to a constant. If it has increased, however, we have gotten further away. This means that the last derivative was the closest derivative to reduce the function to a constant and is therefore the one that I'll use to generate the time complexity. As an example, here is a quadratic sequence but each element has had some noise added to it, with its standard deviation and derivative number in brackets:

{1, 4, 10, 15, 26, 38},  $\sigma = 12.84$  (0)

Here is the sequence after one derivative, along with its standard deviation (note that the deviation has decreased, so we perform another derivative):

{3, 6, 5, 11, 12},  $\sigma = 3.50$  (1)

# Time complexity estimator

## *Documented Design*

After one more (The deviation has decreased so we differentiate again):

$$\{3, -1, 6, 1\}, \sigma = 2.59 \quad (2)$$

Finally:

$$\{-4, 7, -5\}, \sigma = 5.44 \quad (3)$$

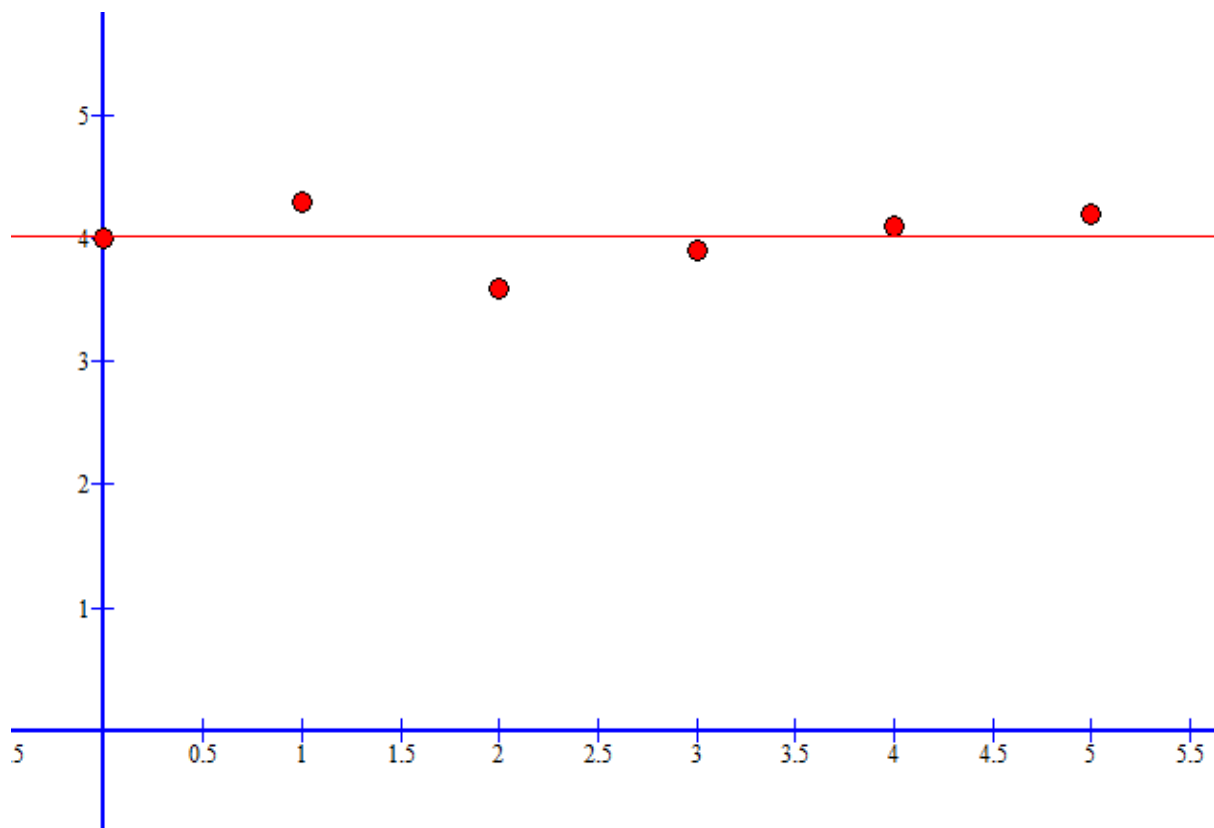
The deviation has increased, meaning that the last derivative was closest to the order of the function, so the function was of order 2, which is true!

### *Algorithm to find the equation of the curve of best fit*

When drawing the graph of a particular function, it is important to be able to see the curve clearly, without having to 'connect the dots' of the nodes mentally. To achieve this, a curve of best fit is required so that a smooth curve can be drawn that most accurately encapsulates the nodes and gives the most realistic prediction of how long other inputs would take. To find this curve I first must have found the time complexity so that I have some information on how the curve must be formed. For each time complexity I will describe how this curve is achieved:

#### Constant

A constant function can be written as  $y = c$ . The line of best fit is the average of the y coordinates of the nodes:



A simple algorithm to find the mean of the times will return the value of  $c$ . This is defined as the sum of all the times divided by the number of nodes there are.

# Time complexity estimator

## *Documented Design*

### Linear

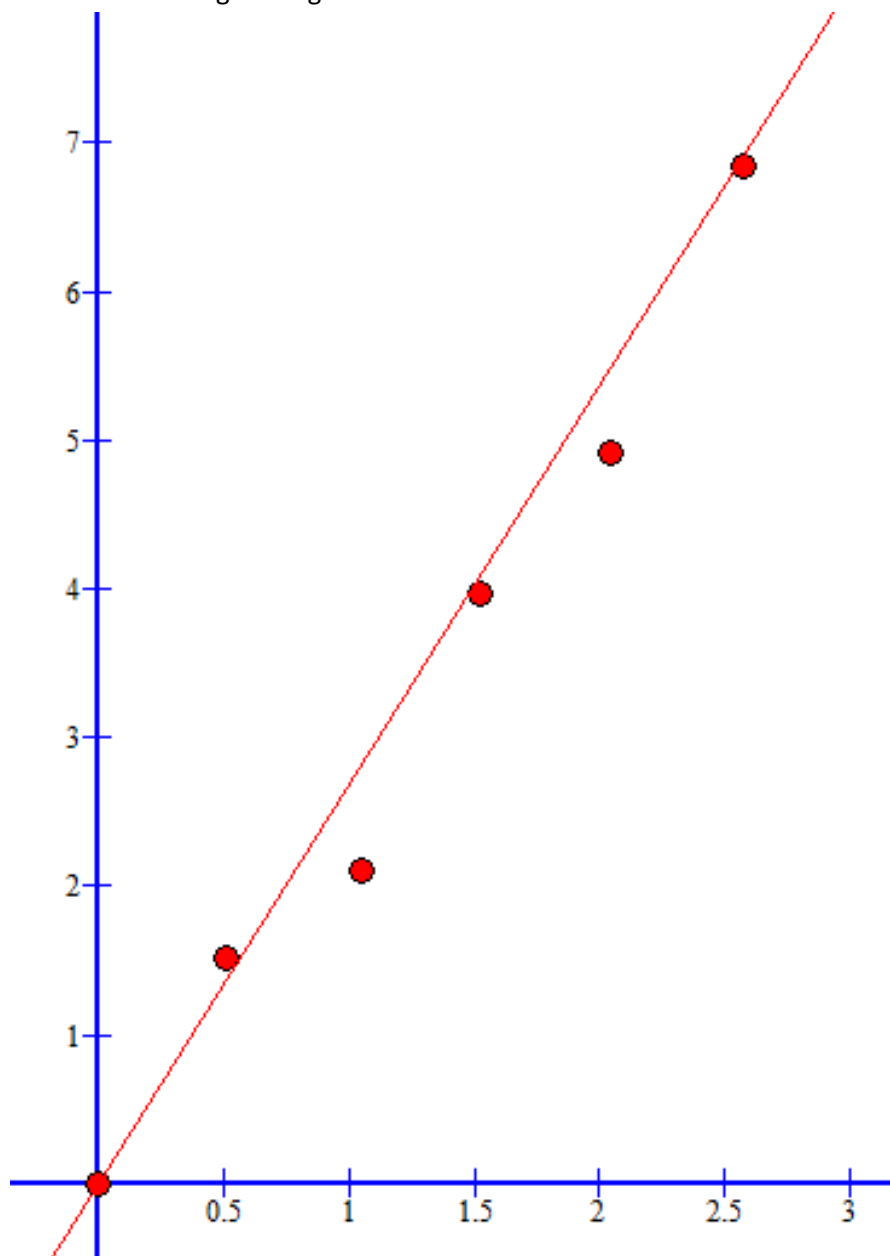
A cartesian line is written as  $y = mx + c$ . The value of 'c' is evidently the time taken for an input of 0. The gradient, or m, can be found by taking the average of the gradients between adjacent points. This can be achieved with the pseudocode:

```
c ← times[0]
totalGradient ← 0;

FOR i ← 1 TO LENGTH(times) DO
    totalGradient := totalGradient + (times[i] - times[i-1]) / (inputs[i] - inputs[i-1])
ENDFOR

gradient ← totalGradient / (LENGTH(times) - 1)
```

The result of using this algorithm is:





# Time complexity estimator

## *Documented Design*

### Logarithmic

The general equation of a logarithmic function is  $y = a \log(x) + c$ . To work out the coefficients 'a' and 'c', the method of linearisation is used. This is the process by which we make the x axis logarithmic, work out the equation of the straight line formed, then substitute the 'x' from  $y = mx + c$  to ' $\log(x)$ '. Both of these processes have already been explained in previous parts of this document.

### Polynomial

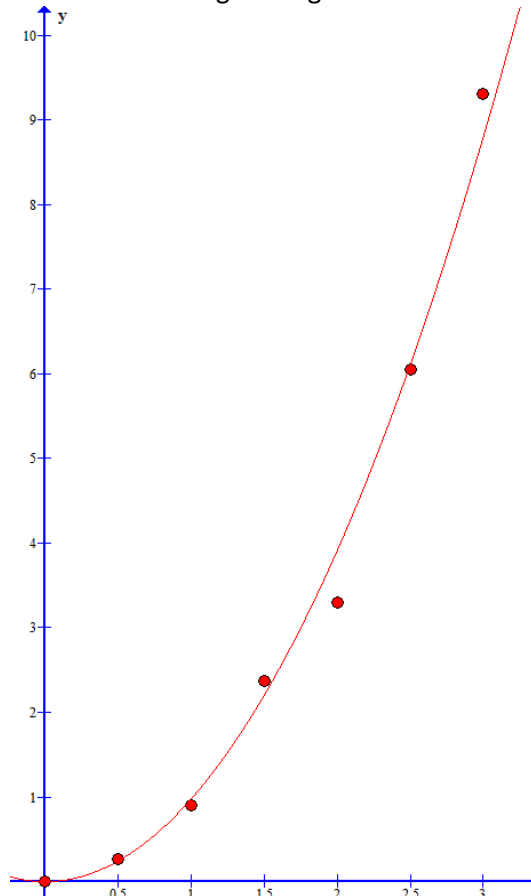
A polynomial curve can be represented by  $y = a x^k + c$ , where 'k' is the order of the polynomial. It is important to note the lack of any terms with the order being less than k, this is because the turning point of the function is always at  $x = 0$ . To find 'c', we use the time for an input of 0. To calculate 'a', we can rearrange the equation to  $a = (y - c) / (x^k)$ . If we calculate the values of 'a' using each point's x and y value, we can take the average of these which will give a good estimate for 'a'.

```
c ← times[0]
totalA ← 0

FOR i ← 1 TO LENGTH(times) DO
    totalA := totalA + (times[i] - c) / (inputs[i] ^ k)
ENDFOR

a ← totalA / (LENGTH(times) - 1)
```

The result of using this algorithm is:



# Time complexity estimator

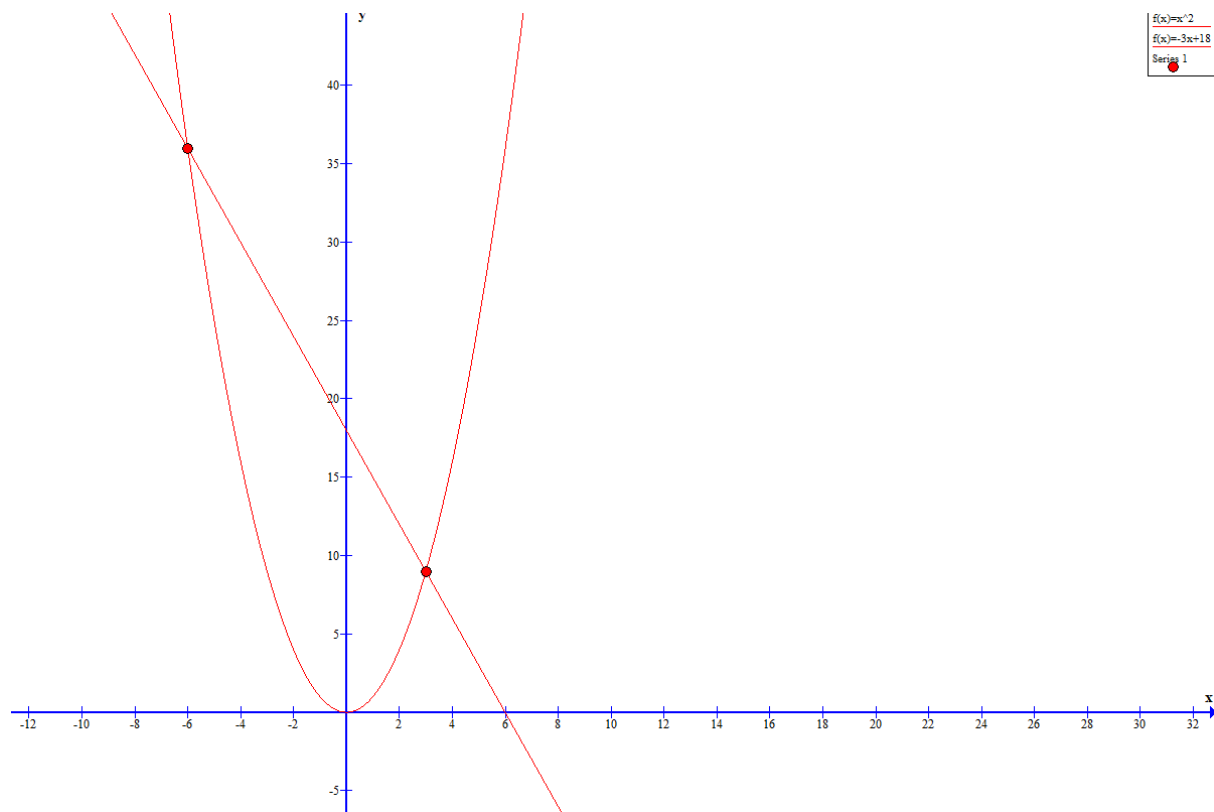
## Documented Design

### Algorithm to find the intersection between two graphs

To calculate intersections of graphs, the cartesian equations are equated and then solved to find the x coordinate of the intersection. For example, to calculate the intersection between the parabola  $y = x^2$  and the line  $y = -3x + 18$ , you can set up an equation:

$$x^2 = -3x + 18$$

And solve it to get  $x = 3$  and  $x = -6$ . This can be verified by looking at the graphs:



However, this is extremely difficult to turn into an algorithmic method that a computer can solve due to the sheer number of cases that must be programmed in, and all the valid operations that can be used. A 'numerical' solution is by far the more practical solution. A numerical solution is one that converges on the answer but will never reach it. It can be seen as a heuristic solution to the problem. The most famous numerical method is called the 'Newton-Raphson method' and can be expressed as follows:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Where  $x$  is the root of the function  $f(x)$ , that being, the solution to  $f(x)=0$ . Whilst the intersection of two graphs does not seem to be of this format, it can be appreciated that:

# Time complexity estimator

## *Documented Design*

$$f(x) = g(x) \Leftrightarrow f(x) - g(x) = 0$$

By subtracting one function from the other, we obtain a function of  $x$  (namely  $f(x) - g(x)$ ) being equal to 0. By applying Newton-Raphson to this function we can obtain  $x$  coordinates for the point of intersection of the two graphs. For this we require the derivatives of each of our functions:

*Derivative of a constant function*

$$\frac{d}{dx}(a) = 0$$

*Derivative of a linear function*

$$\frac{d}{dx}(ax + b) = a$$

*Derivative of a quadratic function*

$$\frac{d}{dx}(ax^2 + c) = 2ax$$

*Derivative of a cubic function*

$$\frac{d}{dx}(ax^3 + c) = 3ax^2$$

*Derivative of a logarithmic function*

$$\frac{d}{dx}(\log_a(x)) = \frac{1}{x \ln(a)}$$

*Derivative of an exponential function*

$$\frac{d}{dx}(a^x) = a^x \ln(a)$$

# Time complexity estimator

## *Documented Design*

It is still difficult to program in the subtraction of the two functions, but it is actually not necessary to combine the two functions into one, since both the numerator and the denominator of the Newton-Raphson formula are associative, meaning that evaluating each function first, and then subtracting the results will yield the same answer as combining the functions and then substituting in values.

The starting value of  $x$  I will use will not be 0 since an intersection at an input of 0 is not a valid solution to the problem. Therefore, a starting value of 1 should result in a valid intersection.

Once a sufficient number of iterations have been calculated, the  $x$  coordinate found is substituted into one of the functions to find the  $y$  value at that point.

The reason a user might want to find the intersection of their graphs is that an intersection indicates when one algorithm performs better than another. If I wrote two algorithms and wanted to know which one would be quicker for an input size of a particular value, finding the intersection and comparing it to that value will tell you the more efficient algorithm.

An implementation of the Newton-Raphson method can be seen here, where  $f$  is a function:

```
FUNCTION NewtonRaphson(f)
    x ← 1

    FOR i ← 0 TO 100 DO
        x := x - f(x) / f'(x)
    ENDFOR

    RETURN x
ENDFUNCTION
```

*Algorithm to determine if a function is exponential*

Given that:

$$\frac{d}{dx}(a^x) = a^x \ln(a)$$

It is clear that repeatedly differentiating an exponential function will always give an exponential function, and it will never become constant. This means that inputting an exponential function to the 'order' pseudocode from above will give a very large number. By checking if this value is greater than, say, 3, we can assume this function is exponential. This does mean that quartic or quintic functions will also be identified as exponential, but it is still a very good approximation.

# Time complexity estimator

## *Documented Design*

### *Simple algorithms to test the program*

To test my program, I will need a simple program for each time complexity:

#### Constant

Any program which is not dependant on the input is constant time complexity.

#### Linear

A program with a single input dependant loop is linear time complexity:

```
FOR i ← 1 TO n DO
    PRINT("Hello World!")
ENDFOR
```

#### Logarithmic

An algorithm that recursively divides the input by two until the result is 1 is logarithmic time complexity:

```
FUNCTION Logarithmic(n)
    IF (n == 1) THEN
        RETURN
    ELSE
        Logarithmic(n / 2)
    ENDIF
ENDFUNCTION
```

#### Quadratic

Embedding an input dependant loop inside another input dependant loop is quadratic time complexity:

```
FOR i ← 1 TO n DO
    FOR j ← 1 TO n DO
        PRINT("Hello World!")
    ENDFOR
ENDFOR
```

# Time complexity estimator

## *Documented Design*

### Cubic

Embedding another input dependant loop inside of the quadratic program will be cubic time complexity:

```
FOR i ← 1 TO n DO
    FOR j ← 1 TO n DO
        FOR k ← 1 TO n DO
            PRINT("Hello World!")
        ENDFOR
    ENDFOR
ENDFOR
```

### Exponential

An algorithm that recursively finds Fibonacci numbers is exponential time complexity:

```
FUNCTION Fibonacci(n)
    IF (n <= 1) THEN
        RETURN n
    END

    RETURN Fibonacci(n - 1) + Fibonacci(n - 2)
ENDFUNCTION
```

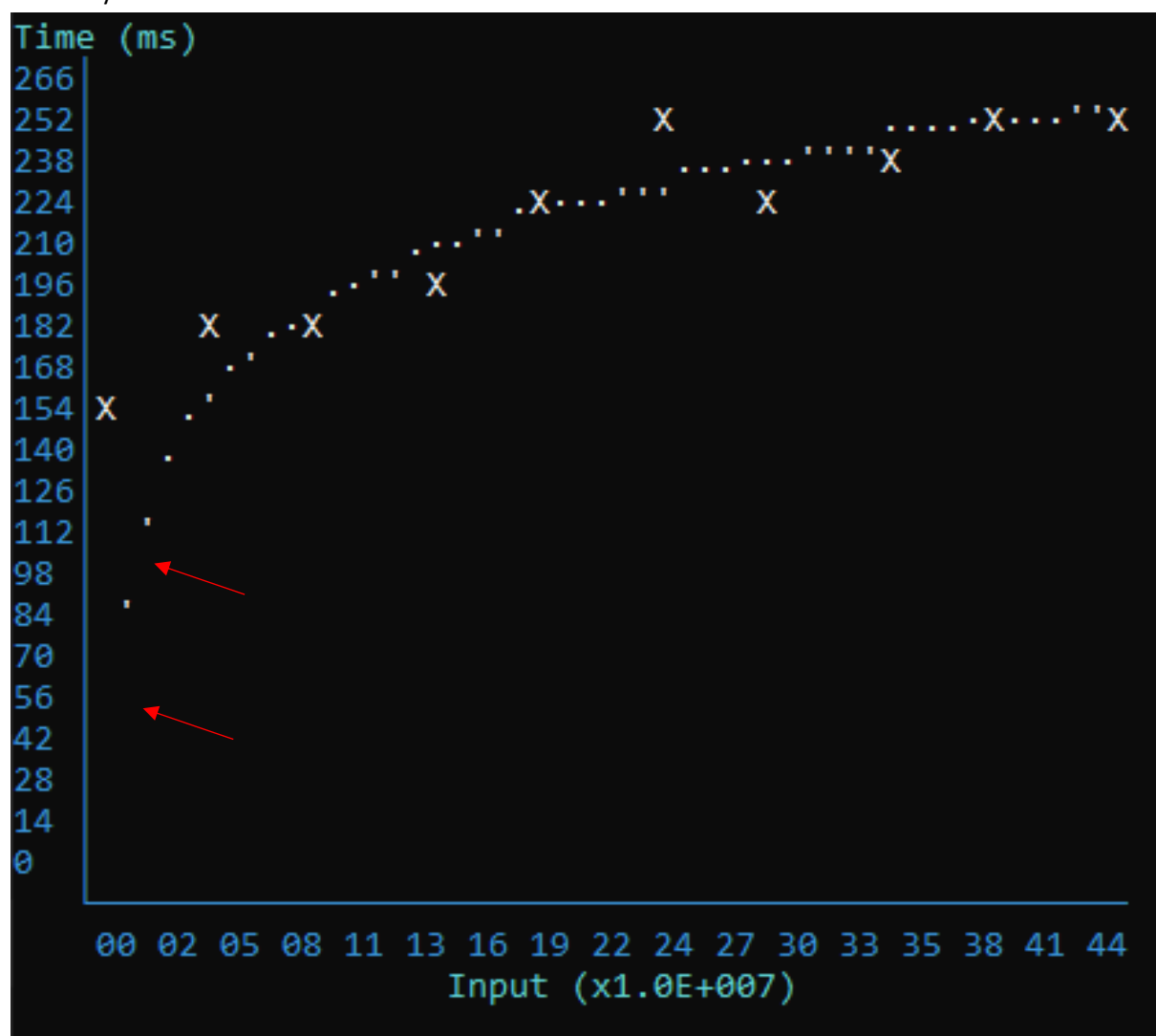
# Time complexity estimator

## Documented Design

### Description of data structures

#### Graph

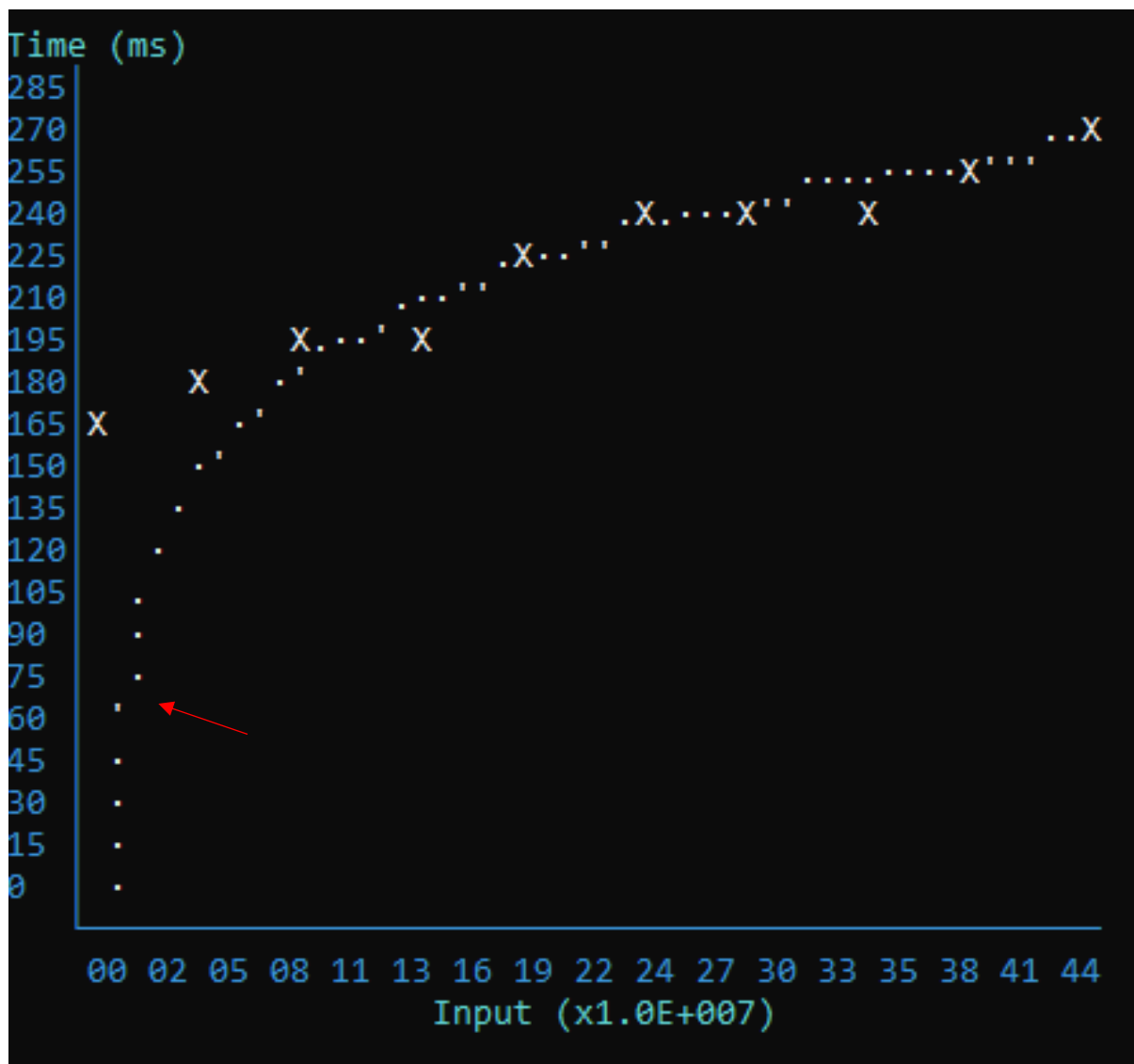
The graph data structure holds a 2D array of Coordinates that can be printed to the screen. To fill this array, there is a method 'SetPoints' which goes through each element and calculates which coordinate should be in that position. To calculate if a node should be in that position, I divide the node x position, its input size, by the x interval on the graph. If this rounded down is equal to the current element's position, then that element is set to a node. The same method is used to find if a given element is on the line of best fit. To make the line of best fit a smooth line, I don't just check if the x coordinate plugged into the generalised function yields the y value. This would make a rough and patchy curve at high gradients, since every x interval could only have one point above it. As an example, this is a graph that only checks the x coordinate for the line of best fit:



The red arrows point to elements which have a gap, due to there being one point for each x value. The solution to this is to also plug in the y values into the inverse function and plot these points along the x direction. Essentially, this is rotating the graph 90 degrees and treating the y axis as the x axis and vice versa. The reason it is the inverse function is because we are reflecting the graph in the line  $y = x$ , therefore making the new function the inverse. This is the result of applying both algorithms, note the extra dots that fill in the gap:

# Time complexity estimator

## Documented Design



The graph data structure also holds the data on the axis. Each axis is held in a 1D array, and each element represents the value that the axis holds for that row/column.

### Algorithm

The algorithm data structure holds all the data on the inputted algorithm, and also contains all the data needed to run it. It has a single method 'Run', which will build and execute the C# file and place all of the results in the 'Results' struct. This data structure does not contain the actual code of the inputted program, only a pointer to where it can be located within the user's file system. However, the algorithm class has been abstracted sufficiently that from outside the class, it behaves exactly as if it contained the program. There are several fields that hold diagnostic information on the progress of running the algorithm. For example, a Boolean field is used to signify whether the program has encountered a fatal error, and another field holds the error that was encountered. For the user to be able to get visual feedback on the progress of the algorithm, the class also has a field which shows how many inputs it has currently ran. Since another field holds how many inputs there are overall, a rough estimate for the percentage completion can be worked out.



# Time complexity estimator

## *Documented Design*

### *Vector*

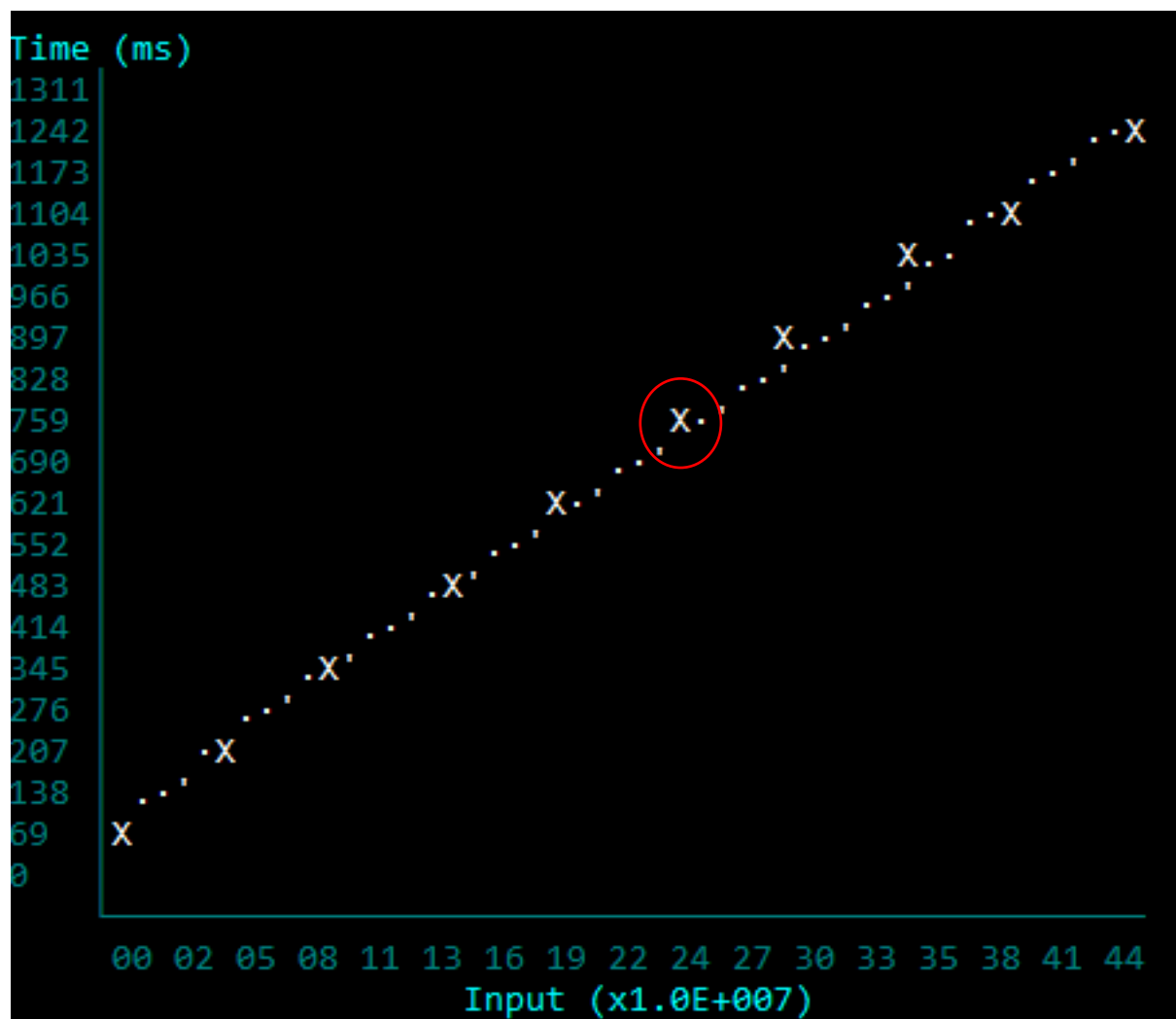
Within the graph, each coordinate needs an x and y value. To improve on readability and maintainability, these two components can be placed in a wrapper called 'Vector', which represents an x and y pair.

### *Coordinate*

The Coordinate data structure holds all the necessary information for every 'cell' in the graph, a cell being a character on the screen. The data it holds is a vector (outlined above) to hold its position in the graph, its colour, and its 'symbol'. Its symbol is the character that is drawn to the screen when the graph is printed out. Since it can be drawn, it implements the 'IDrawable' interface. The coordinate and graph data structures have a compositional relationship since the graph is entirely defined in terms of coordinates.

### *Node*

The node data structure inherits from coordinate. A node is a coordinate which has been run by the program, that is, when running the inputted algorithm these coordinates were used. They are denoted by an 'X' and can be interrogated by the user to get its information. The data the node holds is the input and subsequent time, its colour, whether it is visible on the screen, and whether it is selected. It implements the 'IDrawable' interface as it can be drawn to the screen, as shown below. It has 3 methods, one to draw it and two to display its information. A node is circled below:



# Time complexity estimator

## Documented Design

### Result

Once an algorithm has been processed and data has been gathered on its performance, the result structure is used as a wrapper to keep all the relevant data together. It is defined as a structure since it is immutable, the data will never be changed inside of it. It contains fields for the inputs, times, complexity, and big O.

### Menu








The menu data structure is used to store all the necessary data for an option screen. This includes the text that will appear above the menu and the options to choose from. The options themselves are also data structures, and they are described below. Since a menu can be both drawn and updated, it implements the IDrawable and IUpdateable interfaces













### Menu Option

The menu option structure is used as a wrapper for all of the data for one specific option from a menu. This includes the text and the action that will be executed if it is picked. The structure's purpose is simply to provide an immutable collection of these two pieces of data.

## File design

The '.cs' files are placed by the user in the 'bin' folder of the program, inside of the programs file, this is to separate the user inputted file from the essential files. The C# files are placed inside this folder and once the program has compiled the C# file to an executable it is placed in the same folder.

 Programs	16/11/2021 13:16	File folder	
 Project.deps	04/11/2021 09:01	JSON File	1 KB
 Project.dll	16/11/2021 13:16	Application exten...	37 KB
 Project	16/11/2021 13:16	Application	171 KB
 Project.pdb	16/11/2021 13:16	Program Debug D...	27 KB
 Project.runtimeconfig.dev	04/11/2021 09:01	JSON File	1 KB
 Project.runtimeconfig	04/11/2021 09:01	JSON File	1 KB

 constant.cs	05/10/2021 12:16	C# Source File	1 KB
 constant	16/11/2021 13:16	Application	4 KB
 cubic.cs	05/10/2021 12:16	C# Source File	1 KB
 cubic	09/11/2021 12:41	Application	4 KB
 exponential.cs	05/10/2021 12:16	C# Source File	1 KB
 exponential	16/11/2021 12:57	Application	4 KB
 linear.cs	05/10/2021 12:16	C# Source File	1 KB
 linear	16/11/2021 12:17	Application	4 KB
 logarithmic.cs	05/10/2021 12:16	C# Source File	1 KB
 logarithmic	16/11/2021 13:03	Application	4 KB
 quadratic.cs	05/10/2021 12:16	C# Source File	1 KB
 quadratic	16/11/2021 12:49	Application	4 KB

# Time complexity estimator

## *Documented Design*

### Design of User Interface

#### *Main Menu*

```
*** Time Complexity Estimator ***

Please select your choice:
1 - Enter an Algorithm
2 - About this program
3 - Quit the program
```

#### Title

```
*** Time Complexity Estimator ***
```

The title appears throughout the whole program, it provides a buffer between the top of the window and the menus. I have chosen to have a dark cyan colour so that the menu draws the eye before the title. It also allows the separate screens to feel connected since they have a common feature.

#### Menu Title

```
Please select your choice:
```

This title provides instructions of how to continue. It is in the brightest colour on the screen, light cyan, to draw the user's eye.

#### Menu options

```
1 - Enter an Algorithm
2 - About this program
3 - Quit the program
```

The menu options all have their number in front of them to make it clear of how you select them. They are in a simple white which contrast nicely with the black background.

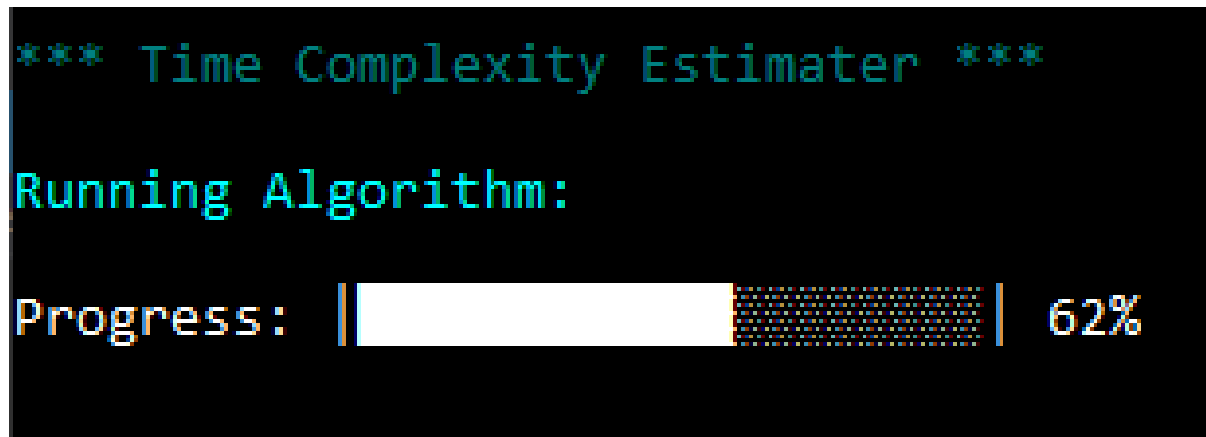
#### *Loading Screens*

```
Running Algorithm:
Calibrating Inputs...
```

# Time complexity estimator

## *Documented Design*

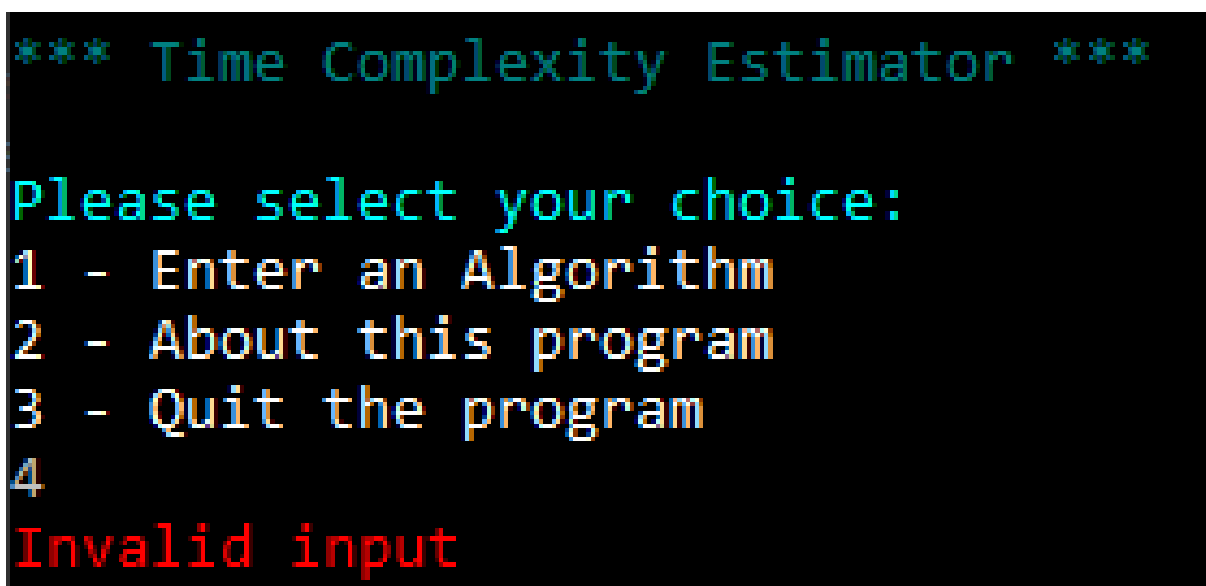
The purpose of a loading screen is to signify to the user that the program is still running even though the user cannot input anything. For this reason, when calibrating inputs, I have added a series of '.'s after the 'Calibrating Inputs' text that counts up to 3 then starts again. This allows the user to be sure that the program has not frozen and provides assurance that the program is still loading. Since calibrating the inputs is not determinate, I cannot add a progress bar, as there is no way to know when it will end.



Once the program has calibrated the inputs, the program will need to run the inputted algorithm with these inputs. Since each individual run of the program with an input of 1 should theoretically be the same, it allows me to get a rough estimate for the percentage completion. This can be approximated as:  $(\text{number of inputs computed so far} / \text{total inputs}) * 100$ . This allows me to create a progress bar so that the user is more likely to wait. Research has shown that a progress bar makes wait times seem shorter than a passive animation<sup>iii</sup>.

## *Errors*

Errors are shown to the user in bright red so that it is immediately obvious that something has gone wrong. The user is then required to press a button to continue. This means that the user has to see the error message before continuing.



# Time complexity estimator

## Documented Design

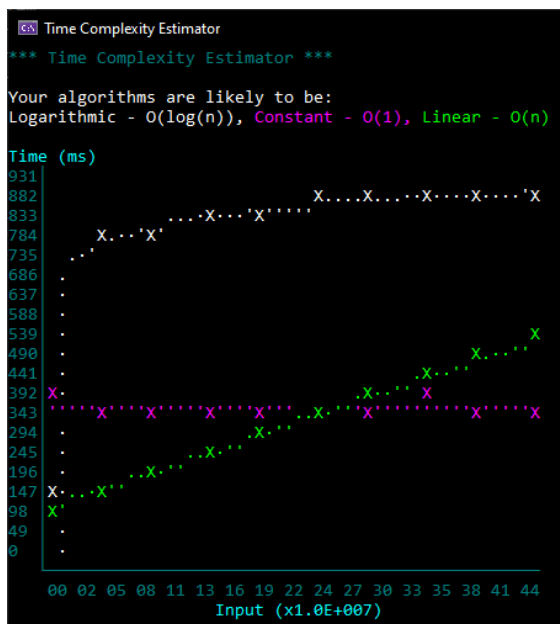
### Graph



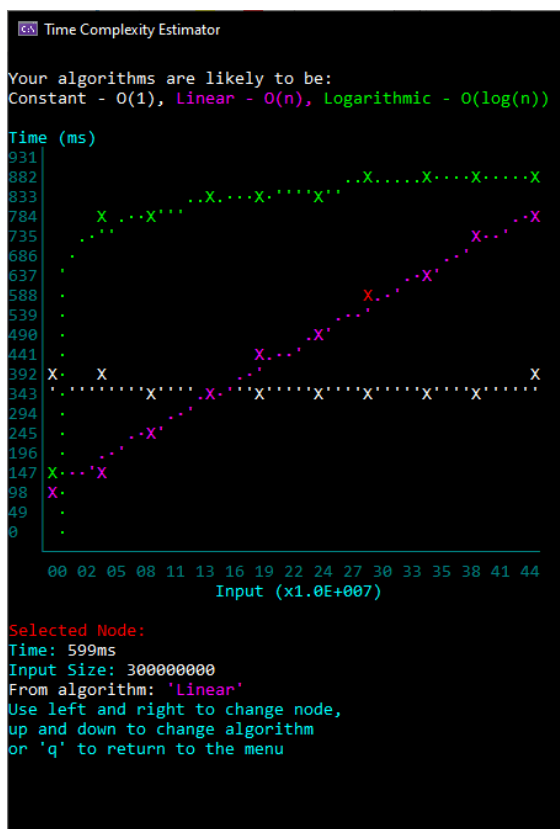
A graph is produced once the algorithm has been ran with the inputs and the times have been recorded from these inputs. The graph uses the same colour scheme as the rest of the program to keep with the theme. The axis labels are a brighter colour, but the points and curves on the graph are white to stand out on the black. The input axis has been labelled with only the first two digits of the number so that it is legible. Underneath is the magnitude of the numbers in standard form so that the axis can be easily read. It also allows the user to easily input numbers when interrogating the graph as the number of zeros is written clearly. I have used crosses for nodes, as detailed above, and dots for the line of best fit markers. These give the appearance of a dotted line, which is conventional when drawing a line of best fit. The axes are made using Unicode lines and corners, as they seamlessly connect together.

# Time complexity estimator

## Documented Design



When multiple results are plotted on the same axes, the different results are colour coded to make it clear as to which plot was identified as which time complexity. I have chosen to use white, purple, green, yellow and grey as the five colours as they are quite distinct. I could not use red as I use red to signify a selected node, as seen below:



I change the selected node to red, and also make the title red so the user makes the connection between the node and the information listed. I also give the colour and time complexity of which result it was from to make it even clearer.

# Time complexity estimator

## Testing

<i>Test Number</i>	<i>Objective being tested</i>	<i>Purpose of test</i>	<i>Section of program (Refer to class diagrams)</i>	<i>Input</i>	<i>Expected result</i>	<i>Actual result</i>	<i>Passed?</i>	<i>Evidence</i>
1	6.1	The user is made aware if invalid data is inputted into the Welcome menu	Welcome	f	Invalid Input	Invalid Input	✓	Screenshot 1
2	6.1	" "	" "	4	" "	" "	✓	Screenshot 2
3	1.3.1	The program will not quit unless the user confirms their choice	" "	3	Are you sure you want to quit?	Are you sure you want to quit?	✓	Screenshot 3
4	2.1.1.1	The program will accept a file with or without a file extension	GetAlgorithm	constant	[Program will run the constant.cs file]	[Program ran constant.cs]	✓	N/A
5	2.1.1.1	" "	" "	constant.cs	" "	" "	✓	N/A
6	2.2.1	The user is made aware if an invalid file has been entered	GetAlgorithm	notAFile.cs	Unable to find the specified file	Unable to find the specified file	✓	Screenshot 4

# Time complexity estimator

## Testing

<i>Test Number</i>	<i>Objective being tested</i>	<i>Purpose of test</i>	<i>Section of program (Refer to class diagrams)</i>	<i>Input</i>	<i>Expected result</i>	<i>Actual result</i>	<i>Passed?</i>	<i>Evidence</i>
7	2.3	The user can return to the main menu if quit is entered	GetAlgorithm	quit	[Program returns to Welcome]	[Program returned to Welcome]	✓	N/A
8	3.3	To check if the user is made aware of an inputted program that doesn't compile.	Algorithm	[A program with an error on line 14]	[An error message describing the problem and line number]	A suitable error [message with line number 14]	✓	Screenshot 5
9	4.2.1 & 4.3	To check if a simple constant program will be correctly identified at 80%	GetTimeComp lexity	[constant.cs * 10]	[The program estimates constant time for at least 8 of them]	[The program estimated 9 constants and 1 linear]	✓	Screenshot 6
10	4.2.2 & 4.3	To check if a simple logarithmic program will be correctly identified at 80%	GetTimeComp lexity	[logarithmic.cs * 10]	[The program estimates logarithmic time for at least 8 of them]	[The program estimated 9 logarithmics and 1 linear]	✓	Screenshot 7
11	4.2.3 & 4.3	To check if a simple linear program will be correctly identified at 80%	GetTimeComp lexity	[linear.cs * 10]	[The program estimates linear time for at least 8 of them]	[The program estimated 10 linears]	✓	Screenshot 8
12	4.2.4 & 4.3	To check if a simple quadratic program will be correctly identified at 80%	GetTimeComp lexity	[Quadratic.cs * 10]	[The program estimates quadratic time for at least 8 of them]	[The program estimated 10 quadratics]	✓	Screenshot 9



# Time complexity estimator

## Testing

<i>Test Number</i>	<i>Objective being tested</i>	<i>Purpose of test</i>	<i>Section of program (Refer to class diagrams)</i>	<i>Input</i>	<i>Expected result</i>	<i>Actual result</i>	<i>Passed?</i>	<i>Evidence</i>
13	4.2.5 & 4.3	To check if a simple cubic program will be correctly identified at 80%	GetTimeComplexity	[cubic.cs * 10] [Max input:] 600000000 [Max time:] 200	[The program estimates cubic time for at least 8 of them]	[The program estimated 10 cubics]	✓	Screenshot 10
14	5.1.1.2	User can rescale the axis to a larger scale	ChangeAxisScale	[Max input:] 600000000 [Max time:] 200	[Axis have been rescaled]	[Axis scaled correctly]	✓	Screenshot 11
15	5.1.1.2	User can rescale the axis to a smaller scale	ChangeAxisScale	[Max input:] 250000000 [Max time:] 140	[Axis have been rescaled]	[Axis scaled correctly]	✓	Screenshot 12
16	5.1.1.3	User can resize the axis to a smaller size	ChangeAxisSize	[X axis length:] 25 [Y axis length:] 15	[Axis have been correctly resized]	[Axis resized correctly]	✓	Screenshot 13
17	5.1.1.3	User can resize the axis to a larger size	ChangeAxisSize	[X axis length:] 100 [Y axis length:] 30	[Axis have been correctly resized]	[Axis resized correctly]	✓	Screenshot 14
18	5.1.1.3	User can resize the X axis to its largest size	ChangeAxisSize	[X axis length:] 120 [Y axis length:] 15	[Axis is at its maximum size]	[The X axis was too large for the screen]	✗	Screenshot 15

# Time complexity estimator

## Testing

<i>Test Number</i>	<i>Objective being tested</i>	<i>Purpose of test</i>	<i>Section of program (Refer to class diagrams)</i>	<i>Input</i>	<i>Expected result</i>	<i>Actual result</i>	<i>Passed?</i>	<i>Evidence</i>
19	5.2.1	The graph can be extrapolated from a time to find the subsequent input	Extrapolate	300	[The estimated input size for a time of 300ms]	10560894303	✓	Screenshot 16
20	5.2.2	The graph can be extrapolated from an input to find the subsequent time	Extrapolate	600000000	[The estimate time for an input of 600000000]	215ms	✓	Screenshot 17
21	5.2.2.	An invalid extrapolation is identified, and the user is warned	Extrapolate	-1	Invalid input	[Nothing was printed but the input was not accepted]	✗	N/A
22	5.3	User can interrogate the graphs and see data on each node	Interrogate	[Left, right, up and down arrow keys]	[The program cycles through nodes and prints out data]	[The program correctly printed out data and moved through nodes]	✓	Screenshot 18
23	5.4.1.1	User cannot add more than 5 graphs to the same axis	DisplayResults	5	A maximum of 5 graphs can be drawn on one axis	A maximum of 5 graphs can be drawn on one axis	✓	Screenshot 19
24	7.1	The user is made aware if they do not have the necessary programs installed.	Program	[Starting the program without the prerequisites]	It seems that Microsoft .NET Framework is not installed, please install and then restart the program	It seems that Microsoft .NET Framework is not installed, please install and then restart the program	✓	Screenshot 20

# Time complexity estimator

## Testing

<i>Test Number</i>	<i>Objective being tested</i>	<i>Purpose of test</i>	<i>Section of program (Refer to class diagrams)</i>	<i>Input</i>	<i>Expected result</i>	<i>Actual result</i>	<i>Passed?</i>	<i>Evidence</i>
25	8.1.1	The user cannot select the intersection option if there is only one graph	DisplayResults	3	There are not enough results for this option	There are not enough results for this option	✓	Screenshot 21
26	8.1.2	The user cannot find the intersection between the same graphs	SelectResult/ DisplayResults	[The linear graph]	[The option for the linear graph is removed]	[The option for the linear graph was removed]	✓	Screenshot 22
27	8.2.1	If there is no intersection, a suitable error is thrown	Intersection	[Two graphs which do not have an intersection point]	No intersection found	No intersection found	✓	Screenshot 23
28	8	The intersection between graphs can be calculated	Intersection	[Two graphs which have an intersection point]	[The program calculates the time and input for the intersection point]	[The program correctly found the point of intersection]	✓	Screenshot 24
29	9 & 4.3	To check if a simple exponential program will be correctly identified at 80%	GetTimeComp lexity	[exponential.c s * 10]	[The program estimates exponential time for at least 8 of them]	[The program estimated 10 exponentials]	✓	Screenshot 25
30	N/A	To check if a bubble sort is correctly identified	GetTimeComp lexity	[A bubble sort algorithm]	[The program estimates quadratic time]	[The program correctly estimated quadratic]	✓	Screenshot 26

# Time complexity estimator

## *Evidence of Testing*

Screenshot 1

```
C:\> Time Complexity Estimator
*** Time Complexity Estimator ***

Please select your choice:
1 - Enter an Algorithm
2 - About this program
3 - Quit the program
f
Invalid input
```

Screenshot 2

```
C:\> Time Complexity Estimator
*** Time Complexity Estimator ***

Please select your choice:
1 - Enter an Algorithm
2 - About this program
3 - Quit the program
4
Invalid input
```

Screenshot 3

```
C:\> Time Complexity Estimator
*** Time Complexity Estimator ***

Are you sure you want to quit?
1 - Yes
2 - No
```

Screenshot 4

```
C:\> Time Complexity Estimator
*** Time Complexity Estimator ***

Enter the filename of the algorithm or enter 'quit' to go back:
notAFile.cs

Unable to find the specified file.
```

Screenshot 5

```
C:\> Time Complexity Estimator
*** Time Complexity Estimator ***

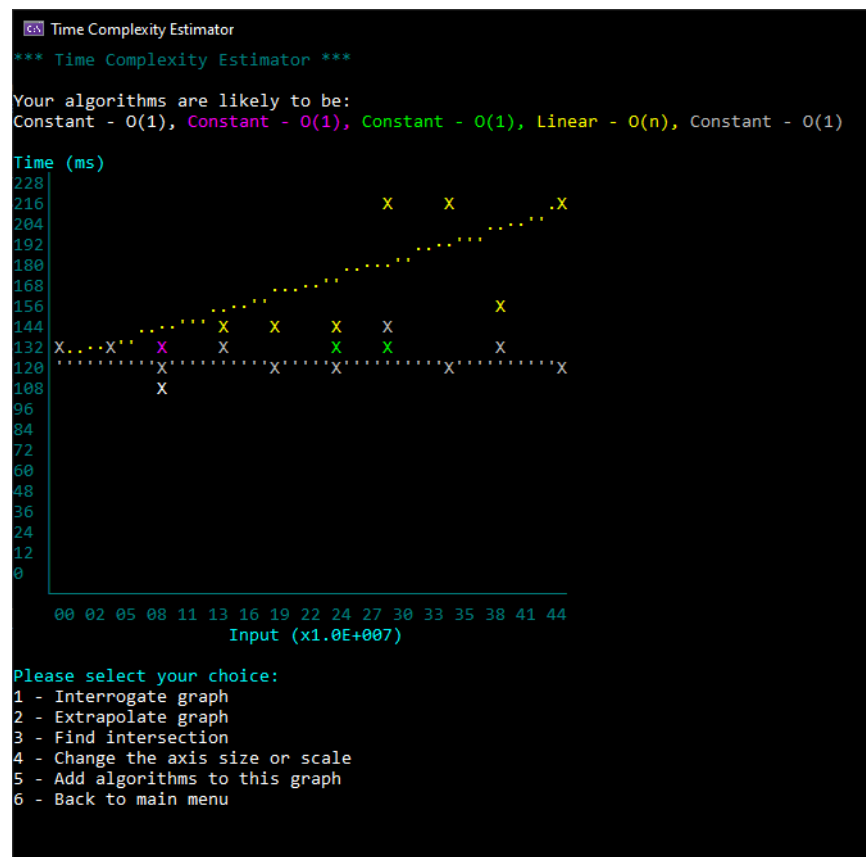
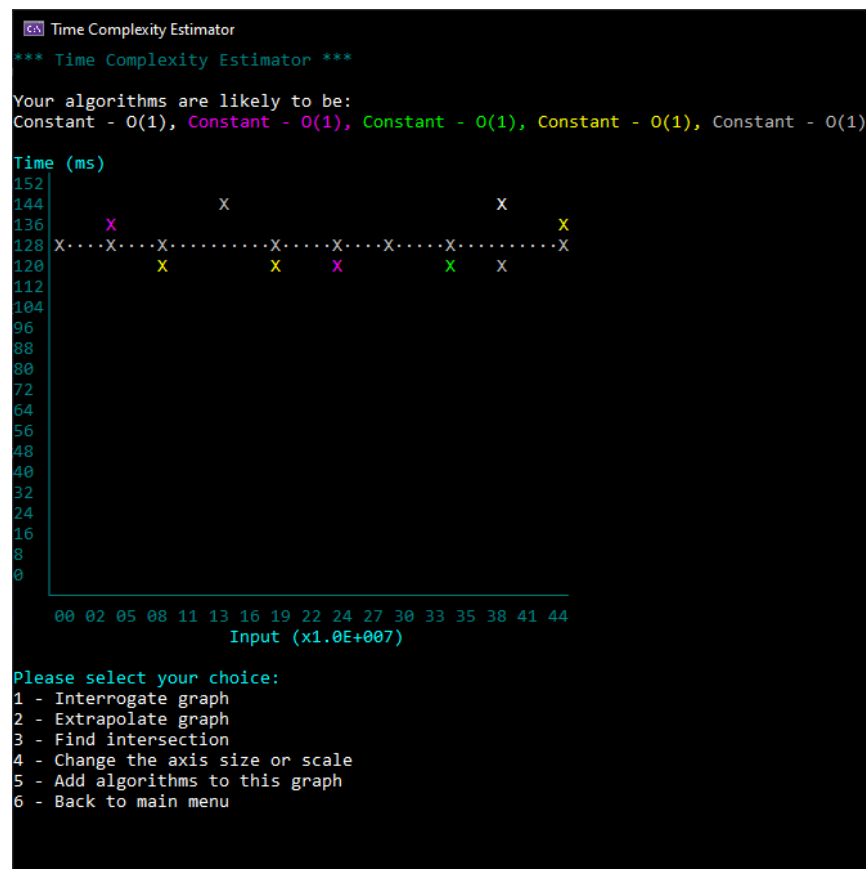
Running Algorithm:
Calibrating Inputs...

An error has occurred, ensure your code compiles correctly. The error was:
constant.cs(14,14): error CS1002: ; expected
```

# Time complexity estimator

## Evidence of Testing

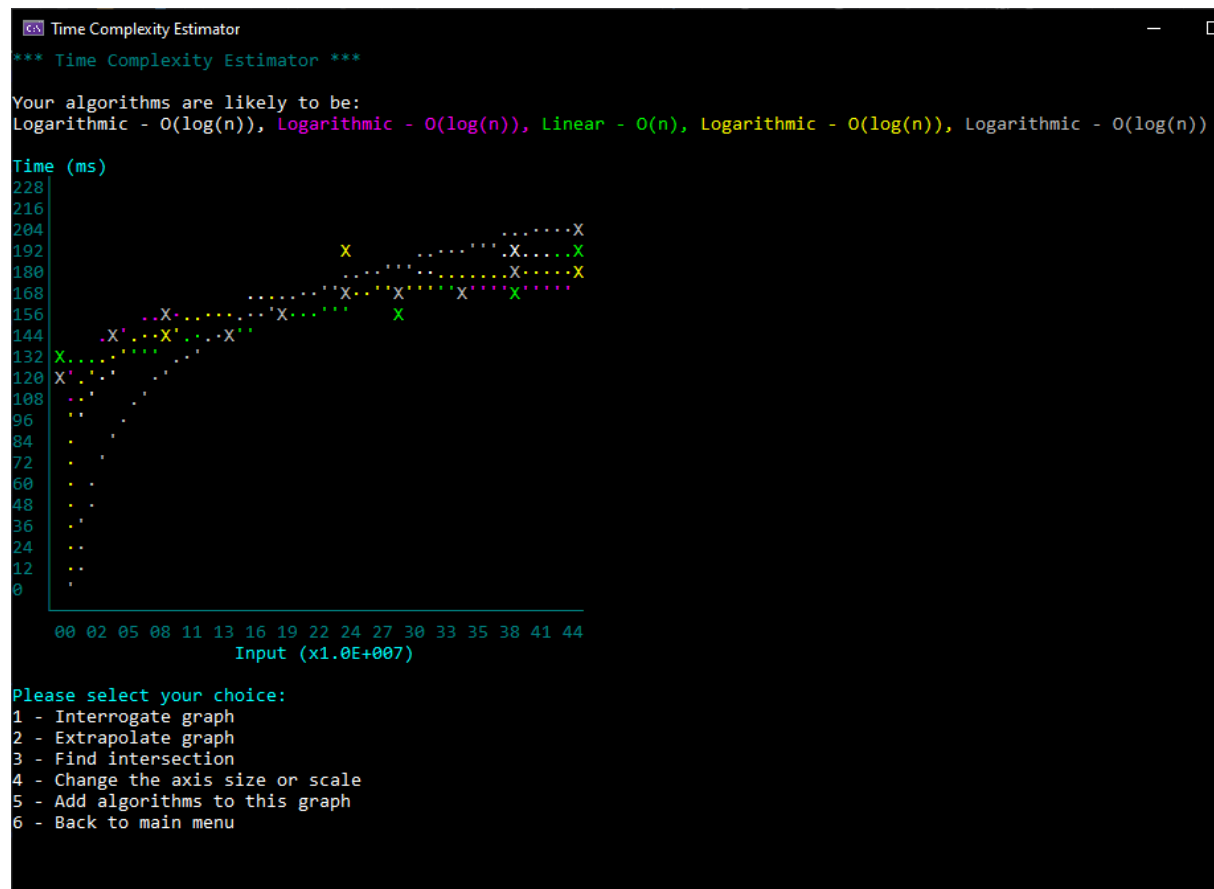
### Screenshot 6



# Time complexity estimator

## Evidence of Testing

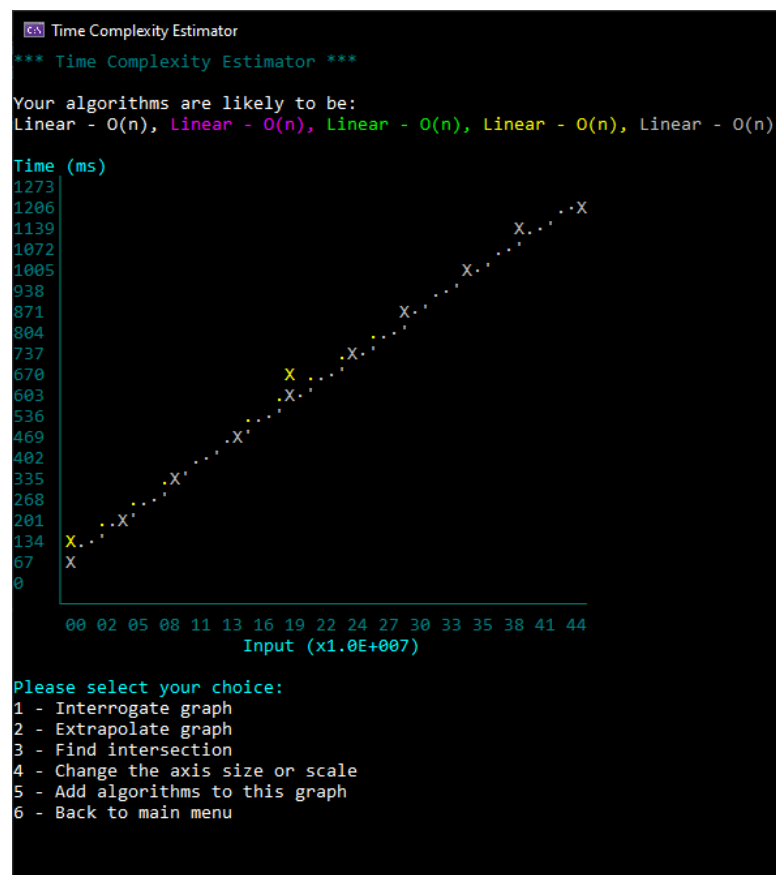
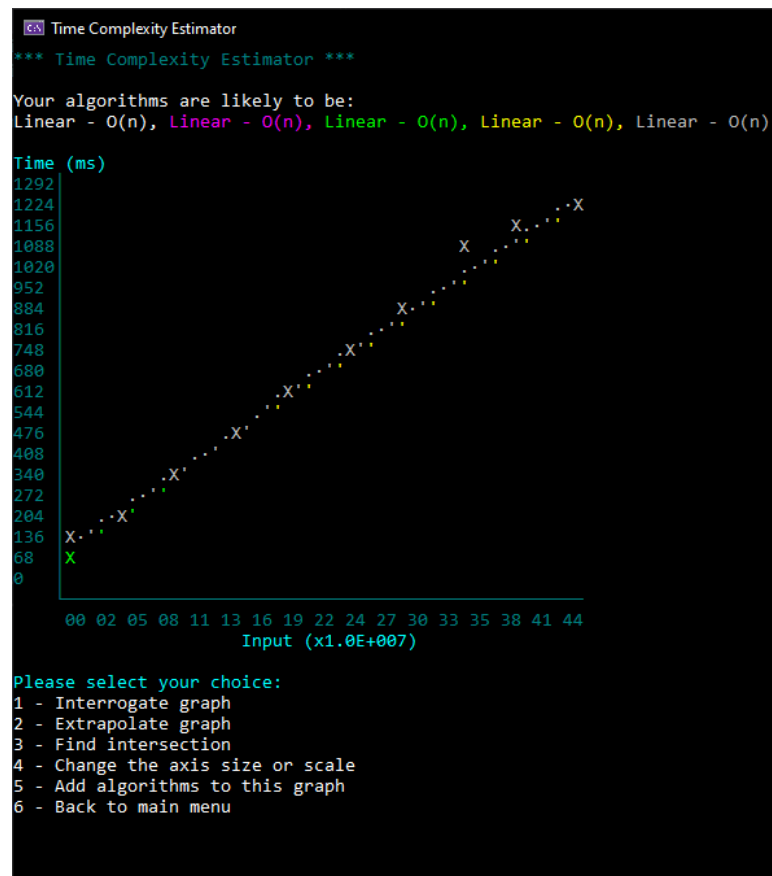
### Screenshot 7



# Time complexity estimator

## Evidence of Testing

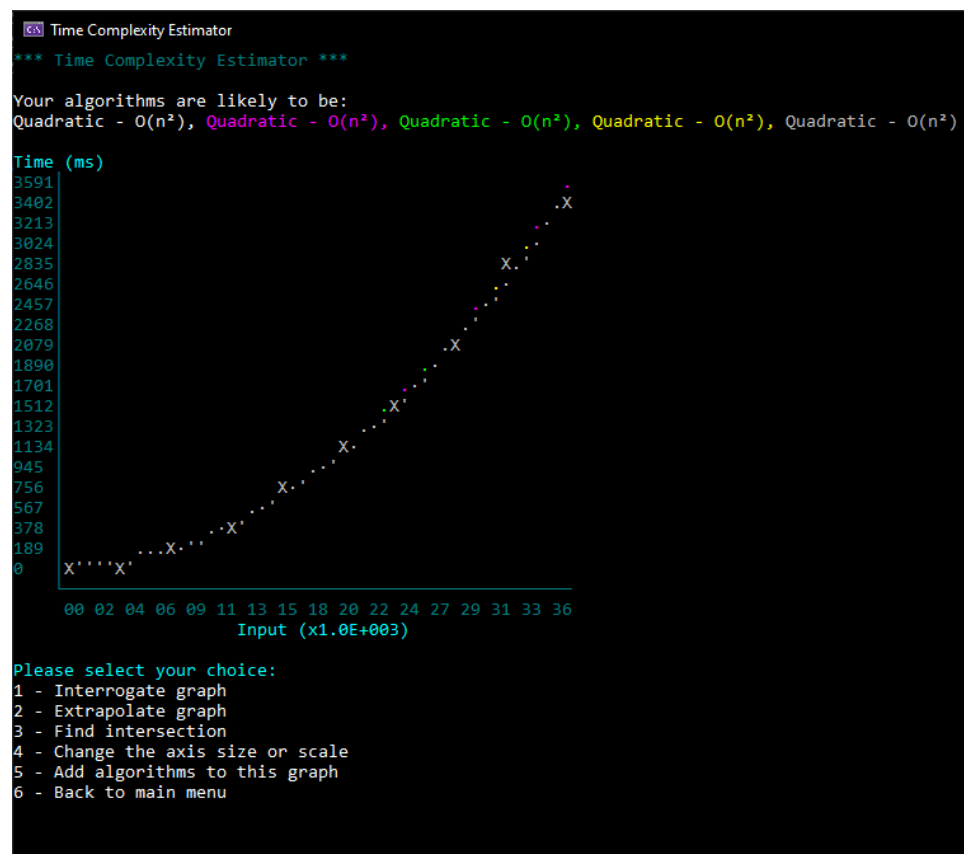
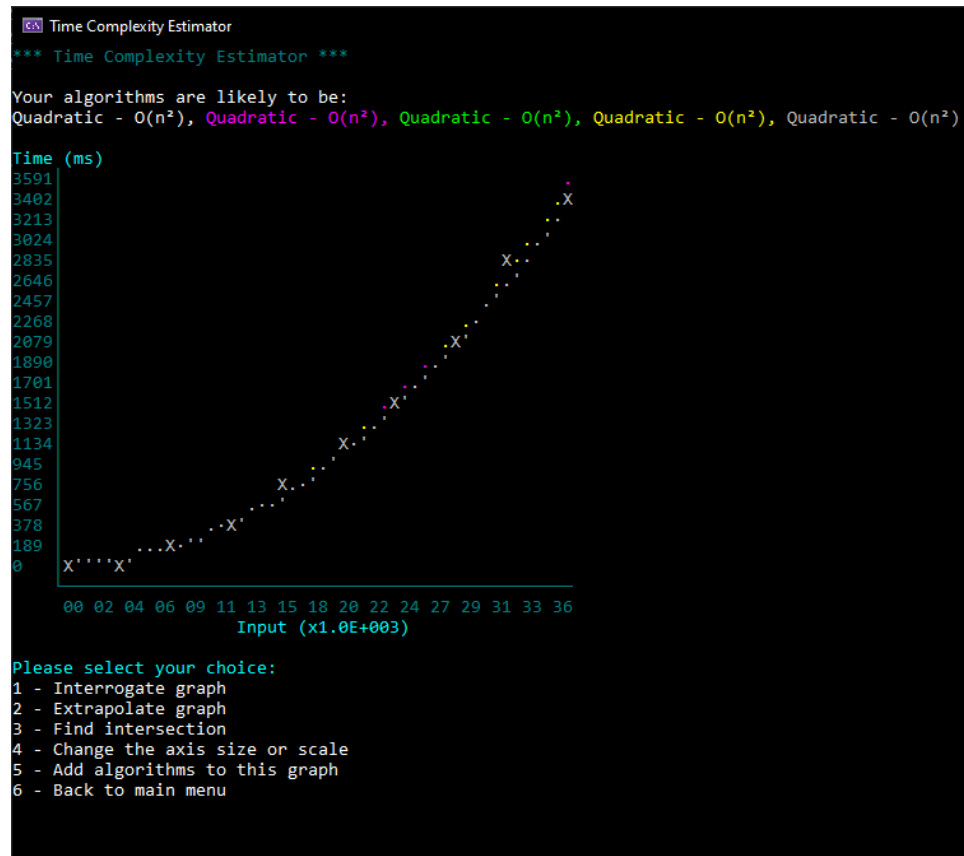
### Screenshot 8



# Time complexity estimator

## Evidence of Testing

### Screenshot 9

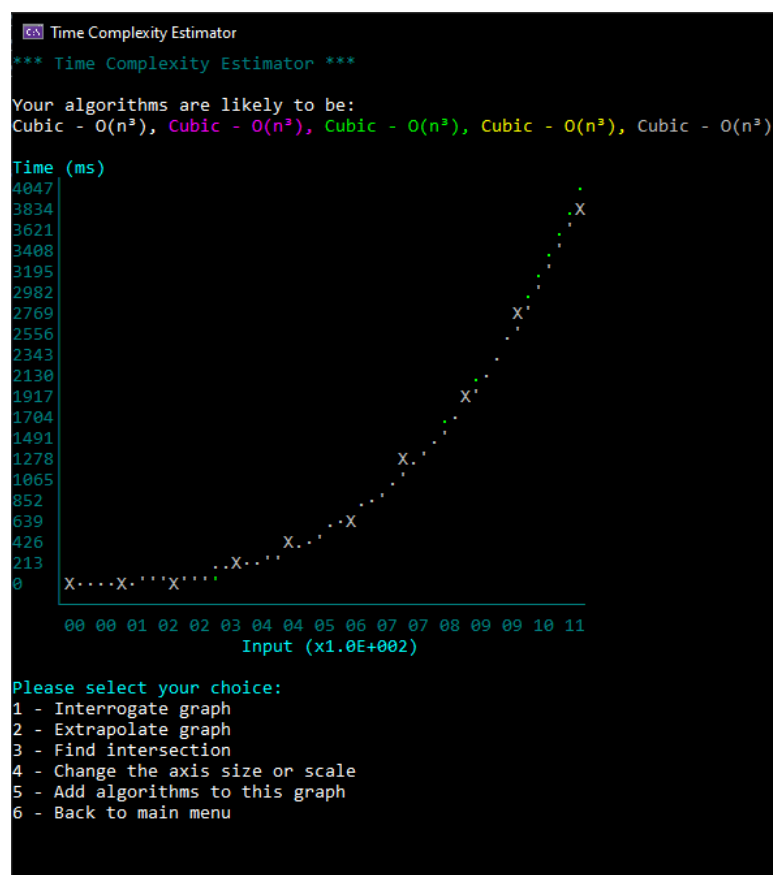
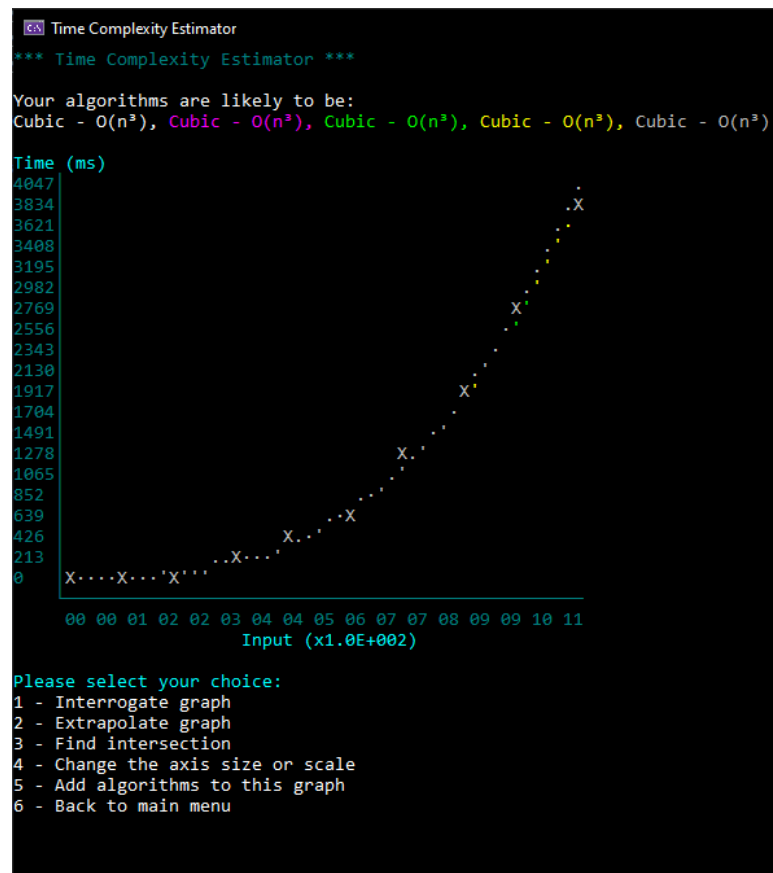




# Time complexity estimator

## *Evidence of Testing*

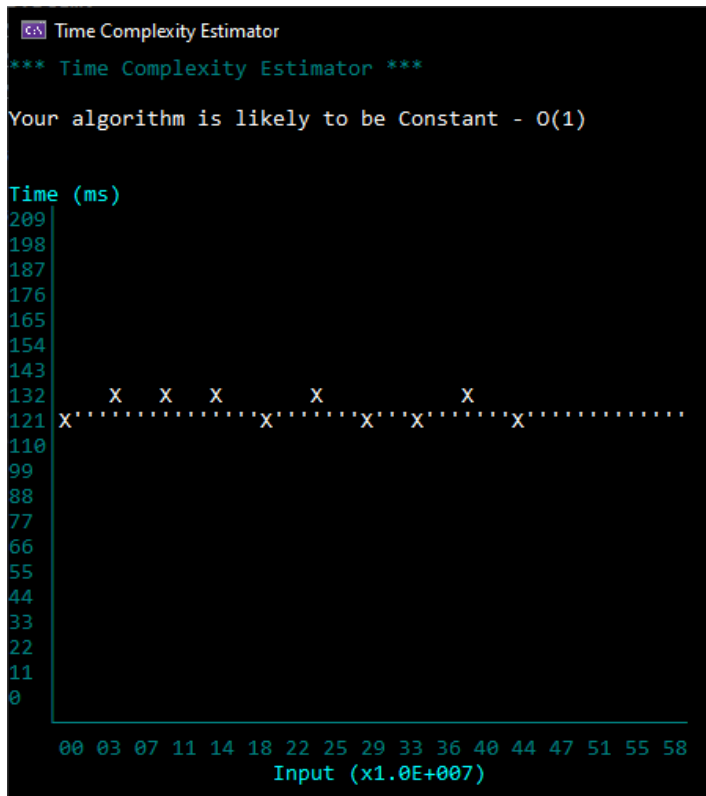
### *Screenshot 10*



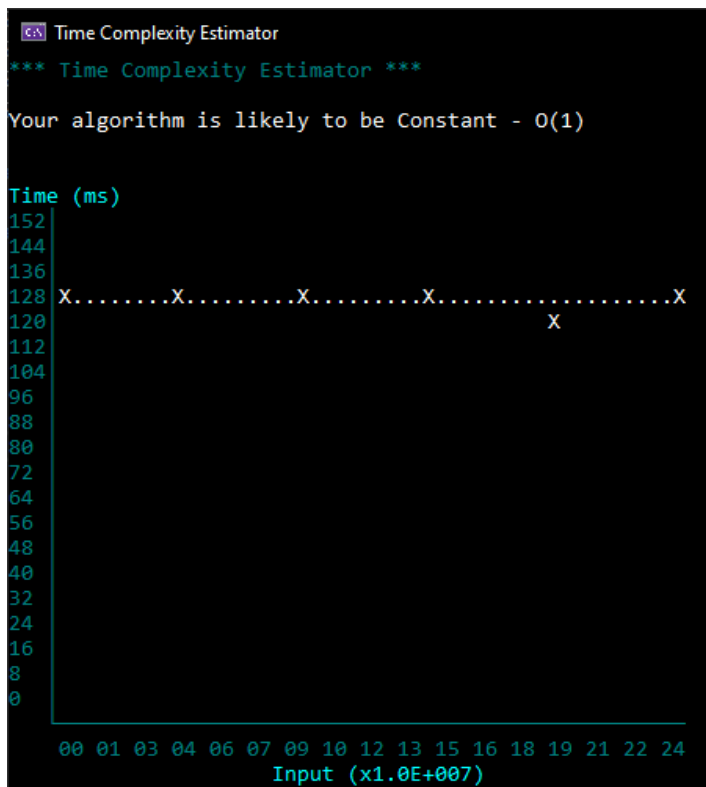
# Time complexity estimator

## Evidence of Testing

Screenshot 11



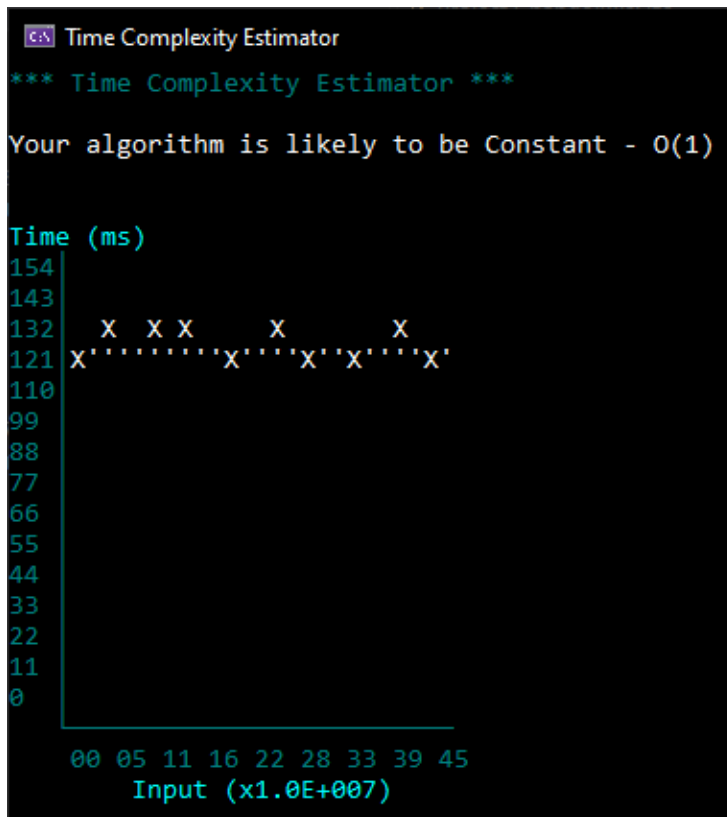
Screenshot 12



# Time complexity estimator

*Evidence of Testing*

Screenshot 13



Screenshot 14



# Time complexity estimator

## *Evidence of Testing*

### *Screenshot 15*

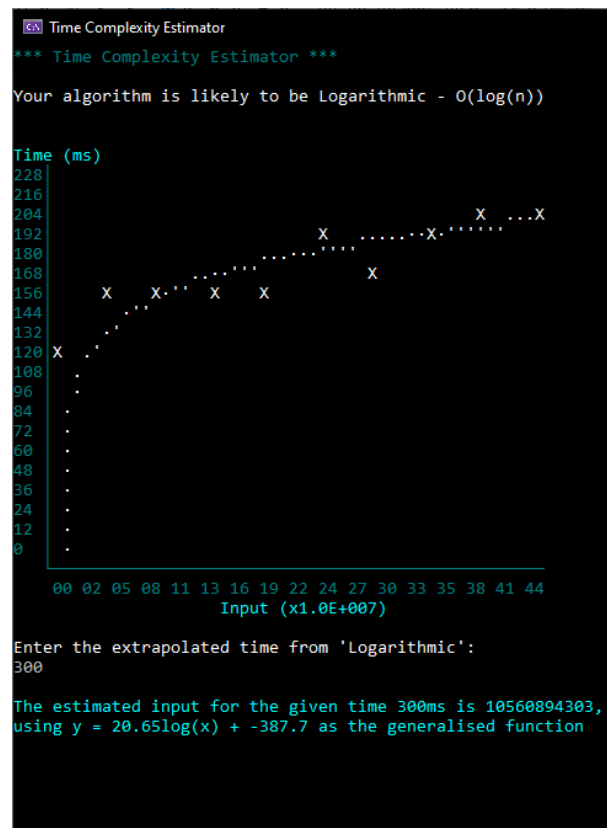


This occurred due to me not factoring in the size of the time axis labels. The size of the actual plot is the size of the width of the screen, but the axis plus the time labels makes it too large.

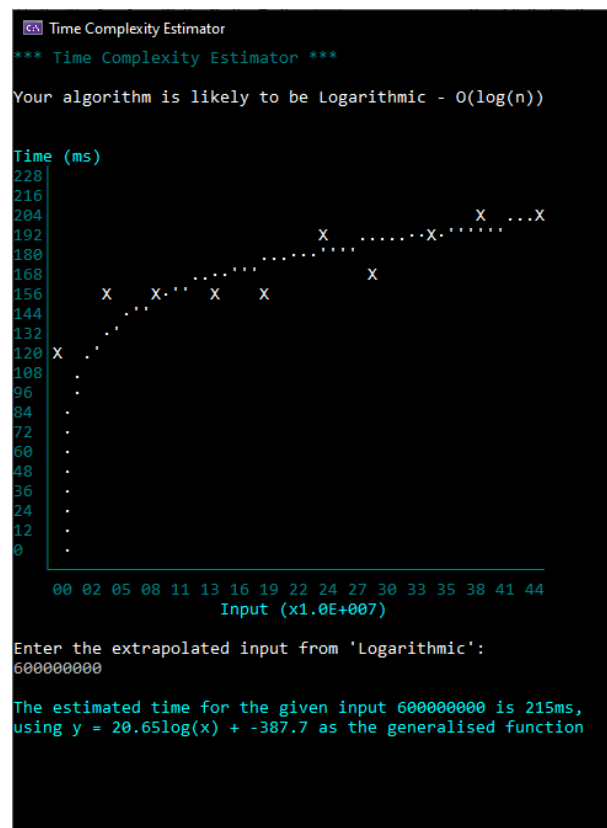
# Time complexity estimator

## Evidence of Testing

Screenshot 16



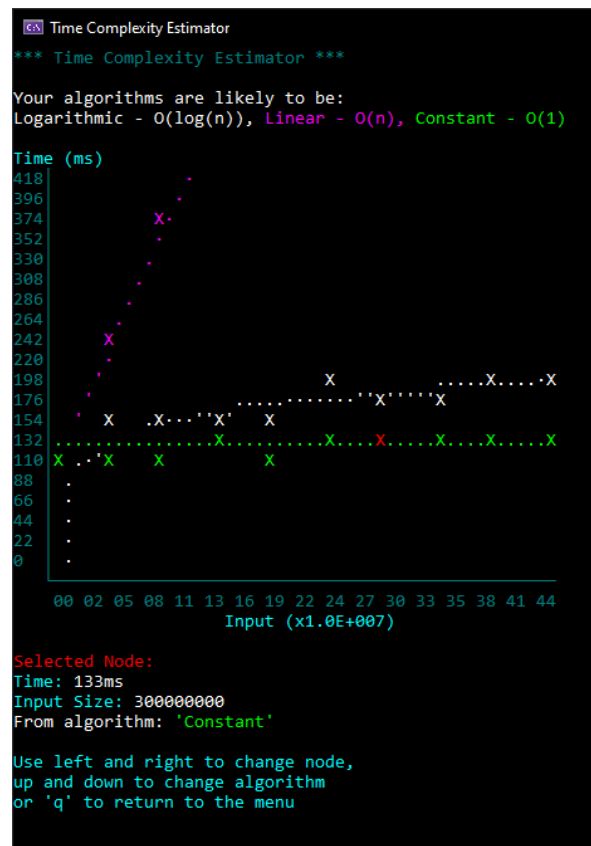
Screenshot 17



# Time complexity estimator

## Evidence of Testing

Screenshot 18



Screenshot 19



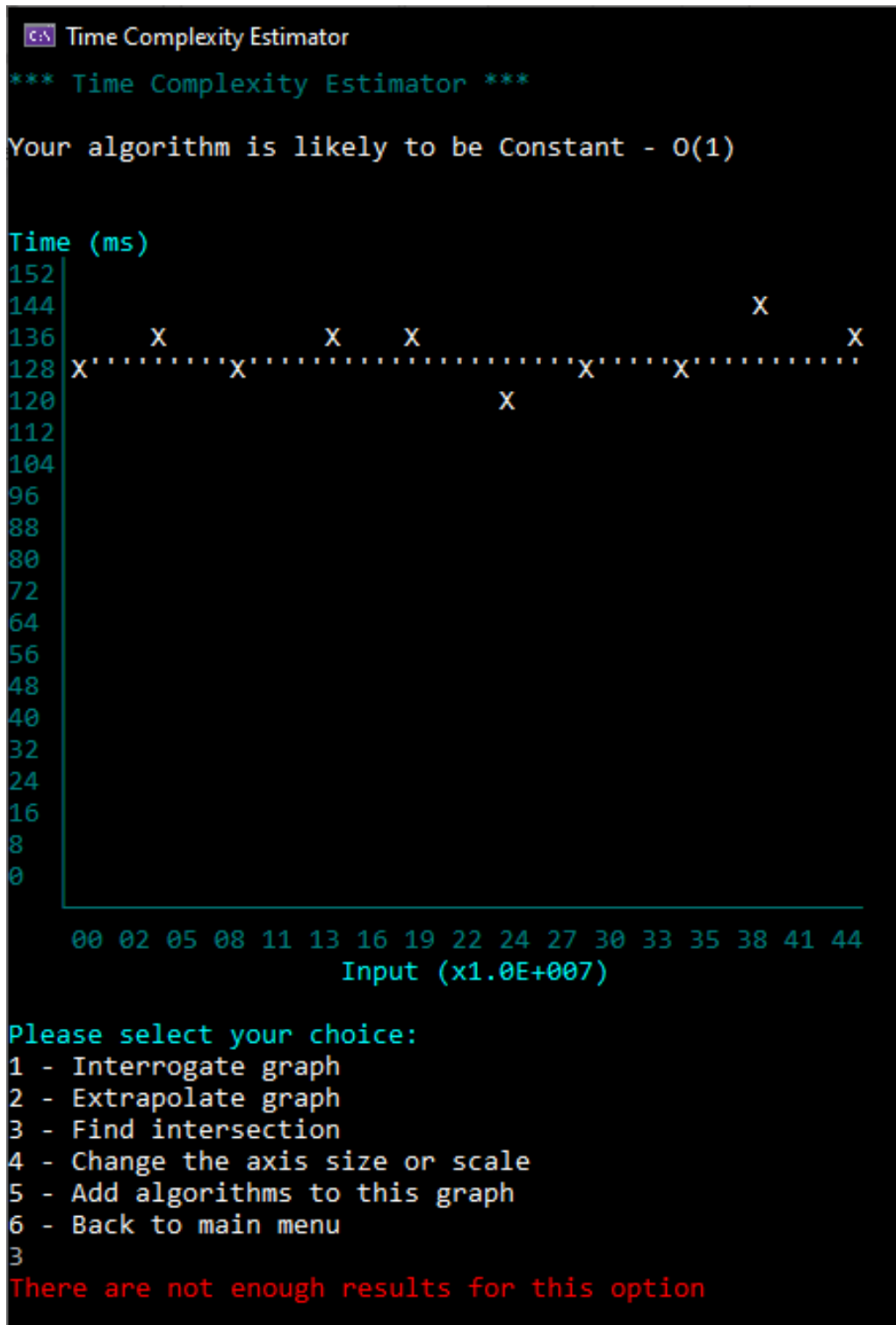
# Time complexity estimator

## Evidence of Testing

Screenshot 20

```
Microsoft Visual Studio Debug Console
It seems that Microsoft .NET Framework is not installed, please install and then restart the program
C:\Users\iling_6z6719y\Documents\Coding\Code\C#\CompProject\Project\bin\Debug\netcoreapp3.1\Project.exe (process 15264)
exited with code -1.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

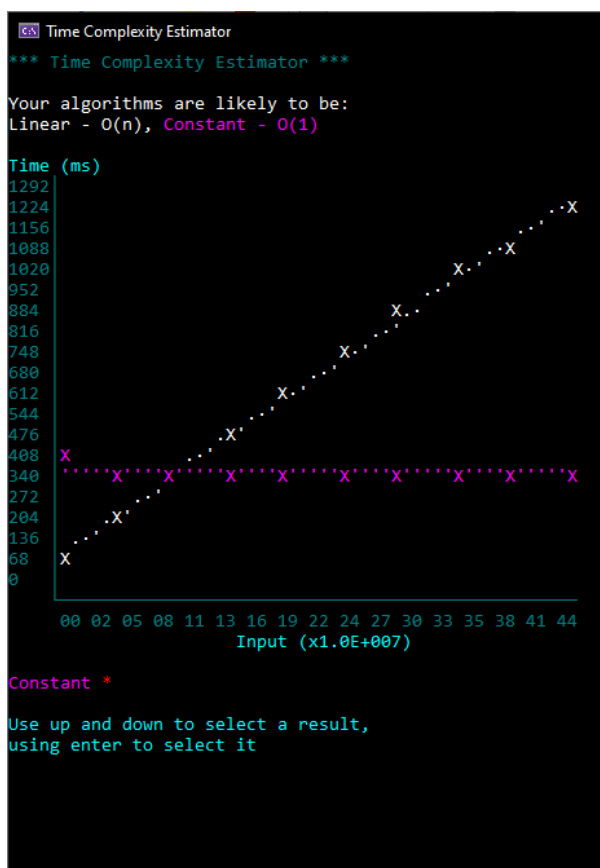
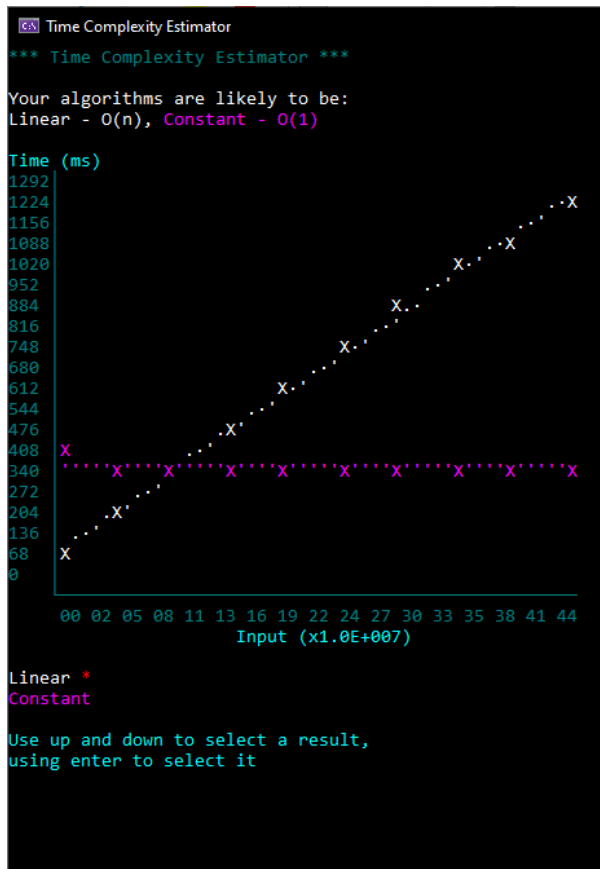
Screenshot 21



# Time complexity estimator

## *Evidence of Testing*

### Screenshot 22

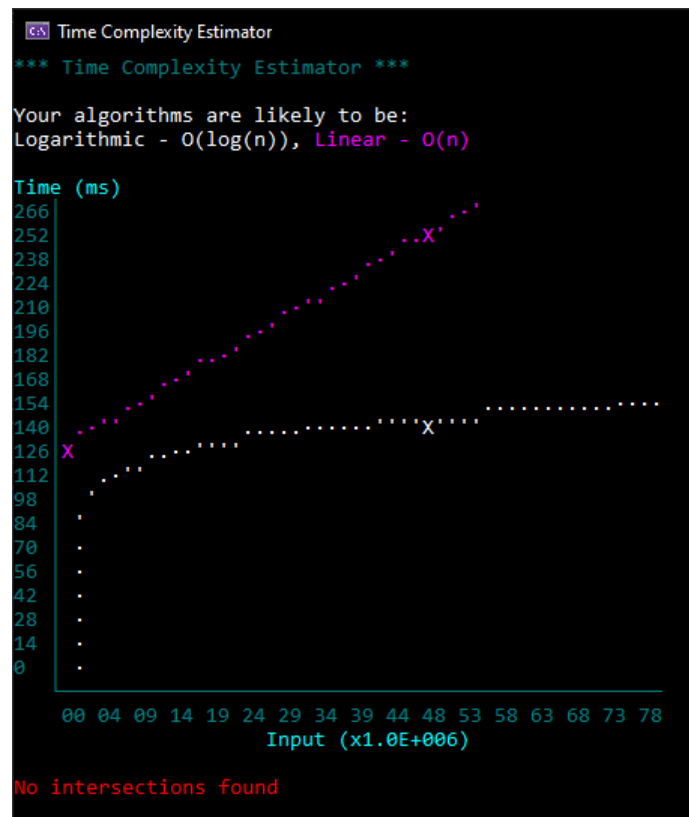




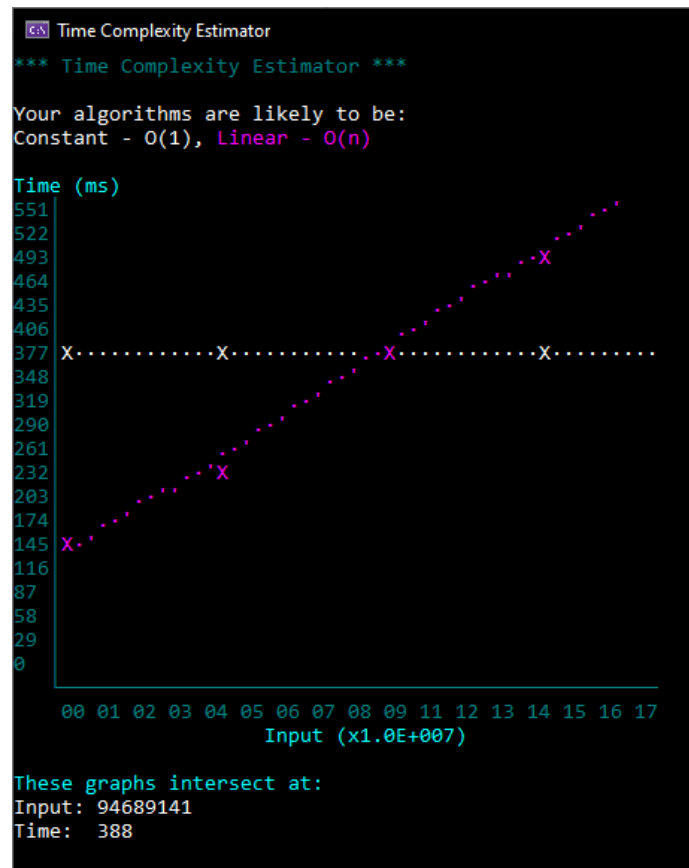
# Time complexity estimator

## Evidence of Testing

Screenshot 23



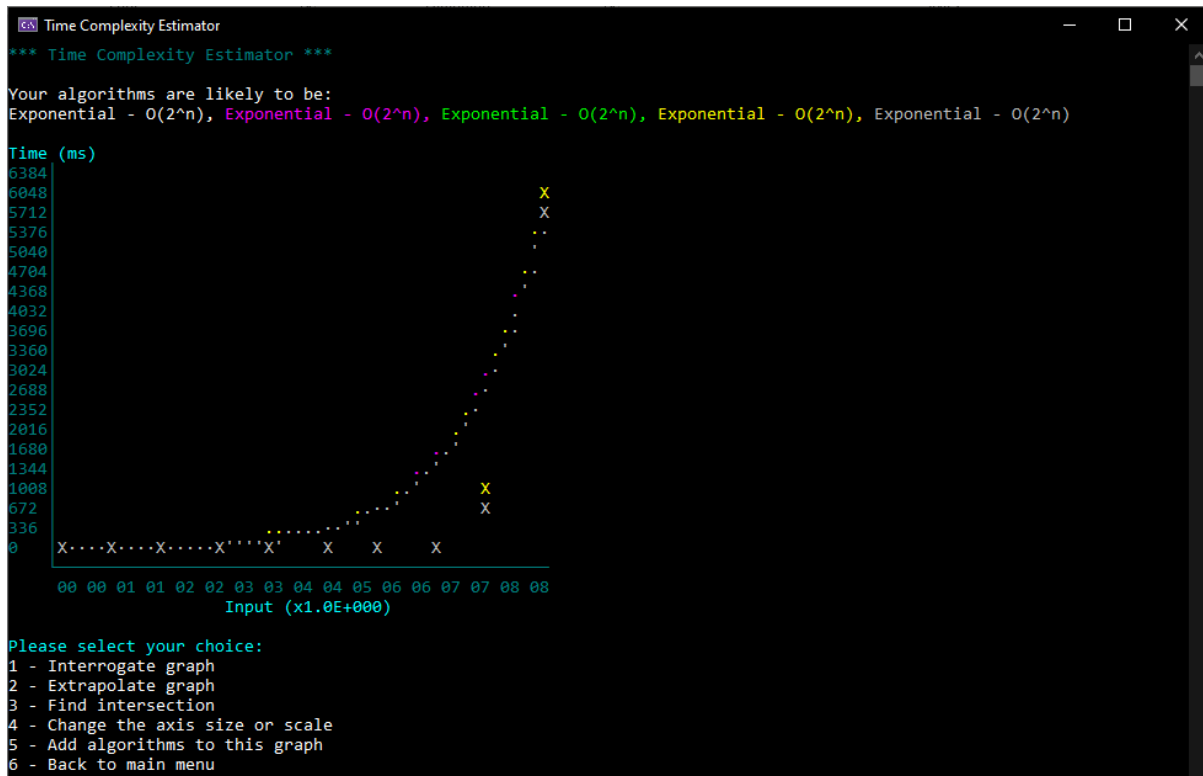
Screenshot 24



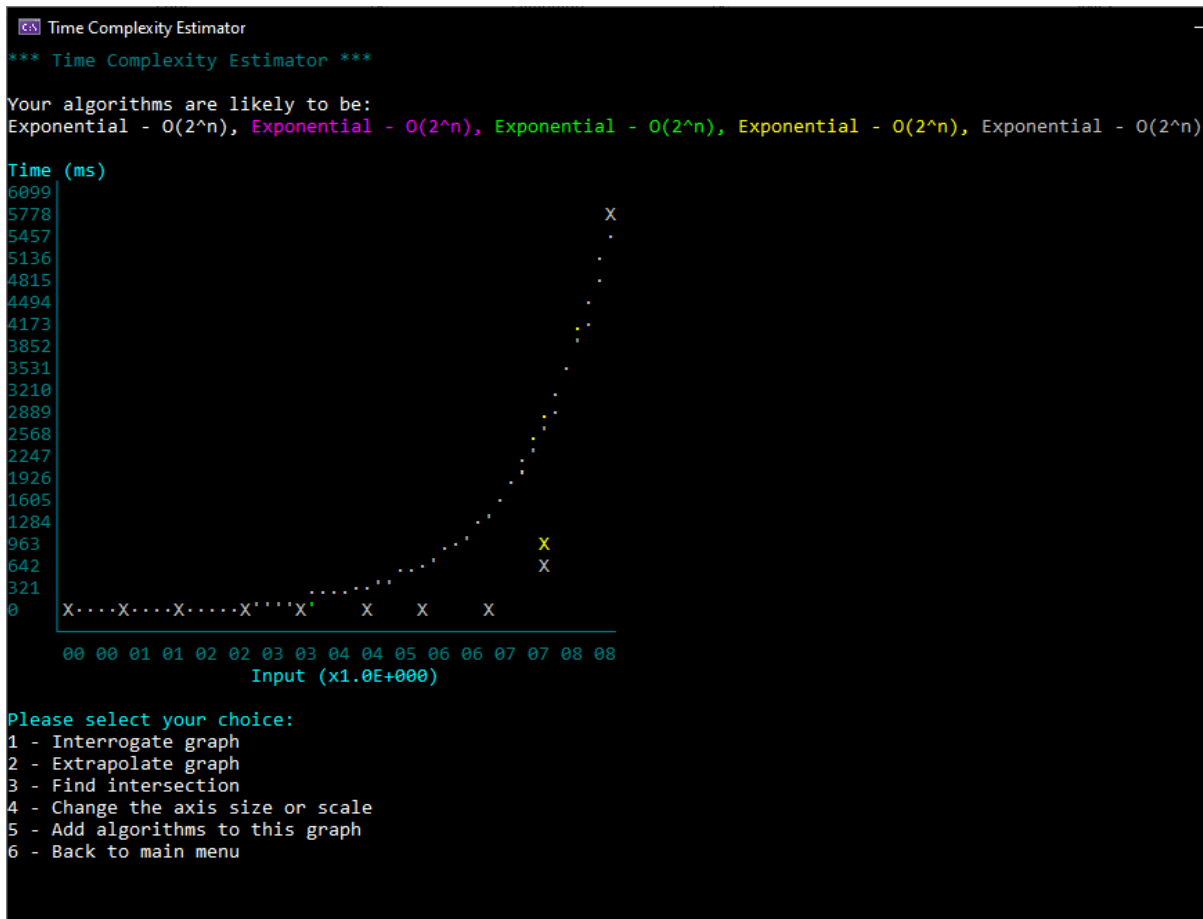
# Time complexity estimator

## Evidence of Testing

Screenshot 25



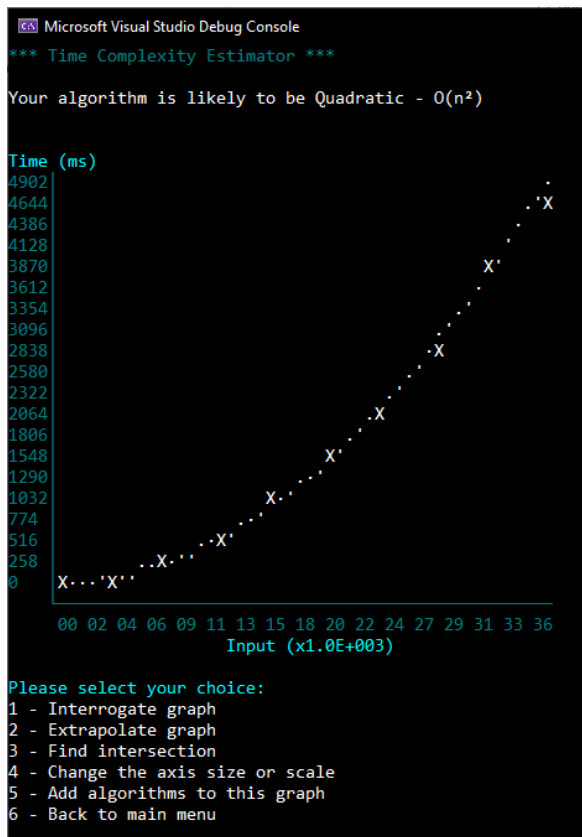
Screenshot 26



# Time complexity estimator

## *Evidence of Testing*

### Screenshot 27



This is the code for the bubble sort:

```
using System;
using System.Diagnostics;
using System.Collections.Generic;
using System.Text;

namespace Program
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                int n = int.Parse(args[0]);
                int[] array = new int[n];

                for (int i = 0; i < n; i++)
                {
                    array[i] = new Random().Next(0, n);
                }

                int temp = array[0];

                for (int i = 0; i < n; i++)
                {
                    for (int j = i + 1; j < n; j++)
                    {
                        if (array[i] > array[j])
                        {
                            temp = array[i];
                            array[i] = array[j];
                            array[j] = temp;
                        }
                    }
                }
            }
            catch (Exception ex)
            {
                Debug.WriteLine(ex);
            }
        }
    }
}
```

# Time complexity estimator

## Technical Solution

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using System.Threading;

namespace TimeComplexityEstimator
{
    static class Program
    {
        static void Main()
        {
            CheckDependencies();

            SetUp();

            StateManager.ChangeState(new Welcome());

            do
            {
                StateManager.DrawCurrentState();
                StateManager.UpdateCurrentState();
            }
            while (!StateManager.ShouldQuit());
        }

        static void SetUp()
        {
            Console.Title = "Time Complexity Estimator";
            Console.WindowHeight = 40;
            Console.CursorVisible = false;
            Directory.SetCurrentDirectory("Programs");
        }

        static void CheckDependencies()
        {
            if (!Directory.Exists("C:\\Windows\\Microsoft.NET\\Framework"))
            {
                Graphics.PrintLine("It seems that Microsoft .NET Framework is not
installed, please install and then restart the program", ConsoleColor.Red);
                Environment.Exit(-1);
            }
        }

        class InvalidInputException : Exception
        {
            public InvalidInputException() : base("Invalid input")
            {
            }
        }

        class GraphLimitReachedException : Exception
        {
            public GraphLimitReachedException() : base("A maximum of 5 graphs can be drawn
on one axis")
            {
            }
        }
    }
}
```

# Time complexity estimator

## Technical Solution

```
class CompilationException : Exception
{
    public CompilationException(string message) : base("An error has occurred,
ensure your code compiles correctly. The error was:\n" + message)
    {
    }
}

class NotEnoughResultsException : Exception
{
    public NotEnoughResultsException() : base("There are not enough results for
this option")
    {
    }
}

/// <summary>
/// This object can be drawn to the screen using the Draw method
/// </summary>
interface IDrawable
{
    public void Draw();
}

/// <summary>
/// This object can be updated using the Update method
/// </summary>
interface IUpdateable
{
    public void Update();
}

class Algorithm
{
    private const int MaxDifference = 50000000;
    private const int Iterations = 10;

    private readonly string fileName;
    private readonly string exeName;
    private int inputSize = 0;
    private int difference = 1;
    private long performedIterations = 0;
    private long totalSize;

    public int PercentComplete { get; private set; }
    public bool IsComplete { get; private set; }
    public Exception ExitException { get; private set; }
    public bool HasCalibratedData { get; private set; }
    public bool Abort { get; private set; }
    public int[] Times { get; private set; }
    public int[] Inputs { get; private set; }

    public Algorithm(string fileName)
    {
        this.fileName = fileName + ".cs";
        exeName = fileName + ".exe";
        Times = new int[Iterations];
        Inputs = new int[Iterations];
    }
}
```

# Time complexity estimator

## Technical Solution

```
    }

    /// <summary>
    /// Builds and runs this algorithm
    /// </summary>
    /// <exception cref="CompilationException">
    /// Thrown when the algorithm fails to build or encounters a fatal error when
running
    /// </exception>
    public void Run()
    {
        try
        {
            // Building the .cs
            using (Process build = new Process())
            {
                build.StartInfo.WindowStyle = ProcessWindowStyle.Hidden;
                build.StartInfo.FileName = "cmd.exe";
                build.StartInfo.RedirectStandardOutput = true;
                build.StartInfo.Arguments = $"/C SET
\"PATH=%PATH%;C:\\Windows\\Microsoft.NET\\Framework\\v4.0.30319\" && cd
\"{Directory.GetCurrentDirectory()}\" && csc \"{fileName}\"";

                build.Start();
                string outputOfBuild = build.StandardOutput.ReadToEnd();

                if (outputOfBuild.Contains("error"))
                {
                    throw new
CompilationException(outputOfBuild.Substring(outputOfBuild.IndexOf(fileName)));
                }
                build.WaitForExit();
                build.Close();
            }

            Stopwatch stopwatch = new Stopwatch();

            // Finding a suitable size of data
            bool isInTime = true;
            do
            {
                difference = Math.Min(difference * 2, MaxDifference);

                using (Process algorithm = new Process())
                {
                    algorithm.StartInfo.WindowStyle = ProcessWindowStyle.Hidden;
                    algorithm.StartInfo.FileName = "cmd.exe";
                    algorithm.StartInfo.RedirectStandardError = true;
                    algorithm.StartInfo.Arguments = $"/C
\"{Path.GetFullPath(exeName)}\" {difference * Iterations}";

                    string error = "";

                    algorithm.Start();
                    isInTime = algorithm.WaitForExit(7500);
                    if (isInTime)
                    {
                        error = algorithm.StandardError.ReadToEnd();
                    }
                    else
                    {
                        // Kill all child processes to avoid memory leaks
                        algorithm.Kill(true);
                        difference /= 2;
                    }
                }
            }
        }
    }
}
```

# Time complexity estimator

## Technical Solution

```
        algorithm.Close();

        if (error != "")
        {
            difference /= 2;
            isInTime = false;
            if (error != "\nUnhandled Exception: OutOfMemoryException.\n")
                throw new Exception(error);
        }
    }

    while (isInTime && difference != MaxDifference);

    totalSize = (long)(0.5f * Iterations * (difference * (Iterations - 1)));

    HasCalibratedData = true;

    // Running the .exe
    for (int j = 0; j < Iterations; j++)
    {
        using (Process algorithm = new Process())
        {
            algorithm.StartInfo.WindowStyle = ProcessWindowStyle.Hidden;
            algorithm.StartInfo.FileName = "cmd.exe";
            algorithm.StartInfo.Arguments = $"/C
\\{Path.GetFullPath(exeName)}\\" {inputSize}";

            // Priming CPU
            for (int k = 0; k < 1000; k++)
            {
                Math.Exp(Math.PI);
            }

            stopwatch.Reset();

            stopwatch.Start();

            algorithm.Start();
            algorithm.WaitForExit();
            stopwatch.Stop();

            algorithm.Close();

            Times[j] = (int)stopwatch.ElapsedMilliseconds;
            Inputs[j] = inputSize;
            performedIterations += inputSize;
            PercentComplete = (int)((float)performedIterations / totalSize *
100);

            PercentComplete = Math.Min(PercentComplete, 100);
            inputSize += difference;
        }

        IsComplete = true;
    }
    catch (Exception ex)
    {
        Abort = true;
        ExitException = ex;
    }
}
```

# Time complexity estimator

## Technical Solution

```
static class MathHelper
{
    public static double StandardDeviation(int[] nums)
    {
        double meanDeviationSquared = SumOfDeviationsSquared(nums, Mean(nums));

        return Math.Sqrt(meanDeviationSquared / nums.Length);
    }

    private static double SumOfDeviationsSquared(int[] nums, double mean)
    {
        double total = 0;

        for (int i = 0; i < nums.Length; i++)
        {
            total += Math.Pow(nums[i] - mean, 2);
        }

        return total;
    }

    public static double Mean(int[] nums)
    {
        double total = 0;

        for (int i = 0; i < nums.Length; i++)
        {
            total += nums[i];
        }

        return total / nums.Length;
    }
}

class TimeComplexityEstimator
{
    public readonly Algorithm algorithm;
    private readonly Complexity[] complexity = new Complexity[]
    {
        Complexity.Constant,
        Complexity.Linear,
        Complexity.Quadratic,
        Complexity.Cubic,
        Complexity.Logarithmic,
        Complexity.Exponential
    };
    private readonly Dictionary<Complexity, string> complexityToBigO = new
Dictionary<Complexity, string>()
    {
        { Complexity.Constant, "1" },
        { Complexity.Logarithmic, "log(n)" },
        { Complexity.Linear, "n" },
        { Complexity.Quadratic, "n^2" },
        { Complexity.Cubic, "n^3" },
        { Complexity.Exponential, "2^n" },
    }
}
```



# Time complexity estimator

## Technical Solution

```
};

public Result ResultOfAlgorithm { get; private set; }
public bool IsComplete { get; private set; }

public TimeComplexityEstimator(Algorithm algorithm)
{
    this.algorithm = algorithm;
}

/// <summary>
/// Will gather results on the inputted algorithm and will put the results in the
ResultOfAlgorithm field
/// </summary>
public void Calculate()
{
    int order = Order(algorithm.Times);

    if (IsConstant(order))
    {
        ResultOfAlgorithm = new Result(Complexity.Constant,
complexityToBigO[Complexity.Constant], algorithm.Times, algorithm.Inputs);
    }
    else if (IsExponential(order))
    {
        ResultOfAlgorithm = new Result(Complexity.Exponential,
complexityToBigO[Complexity.Exponential], algorithm.Times, algorithm.Inputs);
    }
    else if (IsLogarithmic(algorithm.Times))
    {
        ResultOfAlgorithm = new Result(Complexity.Logarithmic,
complexityToBigO[Complexity.Logarithmic], algorithm.Times, algorithm.Inputs);
    }
    else
    {
        int depth = Math.Min(order, 3);
        ResultOfAlgorithm = new Result(complexity[depth],
complexityToBigO[complexity[depth]], algorithm.Times, algorithm.Inputs);
    }

    IsComplete = true;
}

private bool IsConstant(int order)
{
    return order == 0;
}

private bool IsLogarithmic(int[] times)
{
    // By raising a constant to the power of the times, a logarithmic function
    // will reduce to a linear function

    float expBase = 1.01f;
    int[] expOfTimes = new int[times.Length];
    times.CopyTo(expOfTimes, 0);

    // Calibrating expBase to avoid overflow

    int result;
    int input = LargestValueInArray(times);

    if ((int)Math.Pow(expBase, input) < 0)
```

# Time complexity estimator

## Technical Solution

```
{
    do
    {
        result = (int)Math.Pow(expBase, input);
        expBase -= 0.001f;

    } while (result < 0);
}

for (int i = 0; i < expOftimes.Length; i++)
{
    expOftimes[i] = (int)Math.Pow(expBase, times[i]);
}

return Order(expOftimes) == 1;
}

private bool IsExponential(int order)
{
    // Since the differences between subsequent y values for an exponential graph
    // are also exponential, the depth would be big compared to other algorithms

    return order > 3;
}

private int LargestValueInArray(int[] times)
{
    int largest = times[0];

    for (int i = 1; i < times.Length; i++)
    {
        if (times[i] > largest)
        {
            largest = times[i];
        }
    }

    return largest;
}

/// <summary>
/// Returns the order of times
/// </summary>
private int Order(int[] times, int depth = 0, double previousDeviation = 0)
{
    double deviation = MathHelper.StandardDeviation(times);
    if (depth == 0)
        previousDeviation = deviation;

    //Base Case
    if (deviation > previousDeviation || times.Length == 1)
    {
        return depth - 1;
    }
    else
    {
        int[] derivativeOftimes = Differentiate(times);
```

# Time complexity estimator

## Technical Solution

```
        depth++;

        previousDeviation = deviation;

        return Order(derivativeOfTimes, depth, previousDeviation);
    }
}

/// <summary>
/// Returns the derivative of times, given that the inputs have a constant
difference
/// </summary>
private int[] Differentiate(int[] times)
{
    int[] derivativeOfTimes = new int[times.Length - 1];

    for (int i = 0; i < times.Length - 1; i++)
    {
        derivativeOfTimes[i] = times[i + 1] - times[i];
    }

    return derivativeOfTimes;
}

/// <summary>
/// Immutable wrapper for the result generated from TimeComplexityEstimator
calculations
/// </summary>
struct Result
{
    public readonly Complexity complexity;
    public readonly string bigO;
    public readonly int[] times;
    public readonly int[] inputs;

    public Result(Complexity complexity, string bigO, int[] times, int[] inputs)
    {
        this.complexity = complexity;
        this.bigO = "O(" + bigO + ")";
        this.times = times;
        this.inputs = inputs;
    }
}

enum Complexity
{
    Constant,
    Logarithmic,
    Linear,
    Quadratic,
    Cubic,
    Exponential,
    Factorial
}

class GraphState : State
{
    protected Graph graph;
```

# Time complexity estimator

## Technical Solution

```
public GraphState(Graph graph)
{
    this.graph = graph;
}

public override void Draw()
{
    ClearBottomOfScreen();
}

public override void Update()
{
}

public void ClearBottomOfScreen()
{
    Console.SetCursorPosition(0, 10 + graph.YAxisLength);
    for (int i = 0; i < 8; i++)
    {
        Console.WriteLine(new string(' ', Console.BufferWidth));
    }
    Console.WriteLine(new string(' ', Console.BufferWidth));
    Console.SetCursorPosition(0, 10 + graph.YAxisLength);
}

/// <summary>
/// Will allow the user to select a result
/// </summary>
/// <returns>
/// The integer index of the selected result
/// </returns>
public int SelectResult(int removeIndex = -1)
{
    SelectResult selectResult = new SelectResult(graph, removeIndex);

    while (!selectResult.HasFinished)
    {
        selectResult.Draw();
        selectResult.Update();
    }

    return selectResult.ResultIndex;
}

abstract class MenuState : State
{
    protected bool ShouldDrawBanner { private get; set; } = true;
    protected Menu menu;

    public MenuState()
    {
    }

    public override void Draw()
    {
    }
}
```

# Time complexity estimator

## Technical Solution

```
        if (ShouldDrawBanner)
        {
            DrawBanner();
        }
        menu.Draw();
    }

    public override void Update()
    {
        base.Update();
        menu.Update();
    }
}

abstract class State : IDrawable, IUpdateable
{
    public State()
    {
    }

    public virtual void Draw()
    {
        DrawBanner();
    }

    public virtual void Update()
    {
    }

    protected void DrawBanner()
    {
        Console.Clear();
        Graphics.PrintLine("*** Time Complexity Estimator ***", ConsoleColor.DarkCyan);
        Console.WriteLine();
    }
}

class Calculating : State
{
    private readonly TimeComplexityEstimator timeComplexityEstimator;
    private int count = 0;

    public Calculating(TimeComplexityEstimator timeComplexityEstimator)
    {
        this.timeComplexityEstimator = timeComplexityEstimator;
        Console.Clear();
        Graphics.PrintLine("*** Time Complexity Estimator ***", ConsoleColor.DarkCyan);
        Console.WriteLine();
        Graphics.Print("Calculating Time Complexity:", ConsoleColor.Cyan);
        Console.WriteLine("\n");
    }

    public override void Draw()
    {
        Graphics.Print($"Loading{new string('.', count)}{new string(' ',
Console.WindowWidth - 7 - count)}\r", ConsoleColor.White);
    }
}
```

# Time complexity estimator

## Technical Solution

```
public override void Update()
{
    base.Update();

    count++;

    count %= 4;

    if (timeComplexityEstimator.IsComplete && count % 4 == 0)
    {
        if (SavedResults.AreResultsSaved())
        {
            SavedResults.AddResult(timeComplexityEstimator.ResultOfAlgorithm);
            StateManager.ChangeState(new DisplayResults(SavedResults.GetResults(),
true, new Graph(SavedResults.GetResults(), SavedResults.XAxisLength,
SavedResults.YAxisLength)));
        }
        else
        {
            StateManager.ChangeState(new DisplayResults(new List<Result> {
timeComplexityEstimator.ResultOfAlgorithm }, true));
        }
    }

    Thread.Sleep(300);
}

class GetAlgorithm : State
{
    public GetAlgorithm()
    {
    }

    public override void Draw()
    {
        base.Draw();
        Graphics.PrintLine("Enter the filename of the algorithm or enter 'quit' to go
back: ", ConsoleColor.Cyan);
    }

    /// <exception cref="FileNotFoundException">
    /// Thrown when the requested file is not found
    /// </exception>
    public override void Update()
    {
        base.Update();

        string fileName;

        try
        {
            fileName = Console.ReadLine();
            if (fileName.ToLower() == "quit")
            {
                if (SavedResults.AreResultsSaved())
                {
                    StateManager.ChangeState(new
DisplayResults(SavedResults.GetResults(), true));
                }
                else
                {

```

# Time complexity estimator

## Technical Solution

```
        StateManager.ReturnToLastState();
    }
}
else
{
    if (fileName.Contains('.'))
    {
        fileName = fileName.Substring(0, fileName.IndexOf("."));
    }
    if (File.Exists(fileName + ".cs"))
    {
        Algorithm algorithm = new Algorithm(fileName);
        StateManager.ChangeState(new Running(algorithm));
        Thread runAlgorithm = new Thread(() => algorithm.Run());
        runAlgorithm.Priority = ThreadPriority.Highest;
        runAlgorithm.Start();
    }
    else
    {
        throw new FileNotFoundException();
    }
}
}
catch (Exception ex)
{
    InvalidInput(ex);
}
}

private void InvalidInput(Exception ex)
{
    Console.WriteLine();
    Graphics.PrintLine(ex.Message, ConsoleColor.Red);
    Console.ReadKey();
}

}

class Running : State
{
    private readonly Algorithm algorithm;
    private int count = 0;

    public Running(Algorithm runningAlgorithm)
    {
        algorithm = runningAlgorithm;
        Console.Clear();
        Graphics.PrintLine("*** Time Complexity Estimator ***", ConsoleColor.DarkCyan);
        Console.WriteLine();
        Graphics.Print("Running Algorithm:", ConsoleColor.Cyan);
        Console.WriteLine("\n");
    }

    public override void Draw()
    {
        if (algorithm.HasCalibratedData)
        {
            Graphics.Print($"Progress: |{new string('█', algorithm.PercentComplete / 5)}{new string('░', 20 - algorithm.PercentComplete / 5)}| {algorithm.PercentComplete}%\r",
                ConsoleColor.White);
        }
        else
        {

```

# Time complexity estimator

## Technical Solution

```
        {
            Graphics.Print($"Calibrating Inputs{new string('.', count)}{new string(' ',
3 - count)}\r", ConsoleColor.White);
        }
    }

    public override void Update()
    {
        base.Update();

        count++;
        count %= 4;

        if (algorithm.IsComplete)
        {
            TimeComplexityEstimator getTimeComplexity = new
TimeComplexityEstimator(algorithm);
            StateManager.ChangeState(new Calculating(getTimeComplexity));
            Thread calculateTimeComplexity = new Thread(() =>
getTimeComplexity.Calculate());
            calculateTimeComplexity.Start();
        }
        if (algorithm.Abort)
        {
            Console.WriteLine("\n");
            Graphics.Print($"\\n{algorithm.ExitException.Message}", ConsoleColor.Red);
            Console.ReadKey();
            StateManager.ChangeState(new GetAlgorithm());
        }

        Thread.Sleep(300);
    }
}

class Quit : MenuState
{
    public Quit()
    {
        menu = new Menu(
            "Are you sure you want to quit?",
            new MenuOption("Yes", QuitProgram),
            new MenuOption("No", ResumeProgram)
        );
    }

    public override void Draw()
    {
        base.Draw();
    }

    public override void Update()
    {
        base.Update();
    }

    public void QuitProgram()
    {
        StateManager.Quit();
    }

    public void ResumeProgram()
```



# Time complexity estimator

## *Technical Solution*

```
{
    StateManager.ReturnToLastState();
}

static class StateManager
{
    private static State currentState;
    private static State lastState;
    private static bool quit;

    public static void ChangeState(State newState)
    {
        lastState = currentState;
        currentState = newState;
    }

    public static void DrawCurrentState()
    {
        currentState.Draw();
    }

    public static void UpdateCurrentState()
    {
        currentState.Update();
    }

    public static void ReturnToLastState()
    {
        currentState = lastState;
    }

    public static void Quit()
    {
        quit = true;
    }

    public static bool ShouldQuit()
    {
        return quit;
    }
}

class Welcome : MenuState
{
    public Welcome()
    {
        menu = new Menu(
            "Please select your choice:",
            new MenuOption("Enter an Algorithm", Continue),
            new MenuOption("About this program", Information),
            new MenuOption("Quit the program", Quit)
        );
    }

    public void Continue()
    {
        StateManager.ChangeState(new GetAlgorithm());
    }
}
```

# Time complexity estimator

## Technical Solution

```
    }

    public void Quit()
    {
        StateManager.ChangeState(new Quit());
    }

    public void Information()
    {
        Console.Clear();
        Graphics.PrintLine("*** Time Complexity Estimator ***", ConsoleColor.DarkCyan);
        Console.WriteLine();
        Graphics.PrintLine("This program will estimate the time complexity of an algorithm in big  
O notation", ConsoleColor.White);
        Graphics.PrintLine("The user must be aware that due to Rice's theorem it is impossible to  
calculate the", ConsoleColor.White);
        Graphics.PrintLine("true time complexity of an algorithm but it can be estimated for  
relatively small", ConsoleColor.White);
        Graphics.PrintLine("programs.", ConsoleColor.White);
        Console.WriteLine();
        Graphics.PrintLine("To use this program, simply place your desired algorithm as a '.cs' in  
the bin", ConsoleColor.White);
        Graphics.PrintLine("folder of this program within the 'Programs' directory. Then select  
option 1 from the", ConsoleColor.White);
        Graphics.PrintLine("menu and input the name of the file. Ensure that there are no CPU  
intensive tasks", ConsoleColor.White);
        Graphics.PrintLine("running whilst the algorithm is running.", ConsoleColor.White);
        Console.WriteLine();
        Graphics.PrintLine("Press enter to continue", ConsoleColor.Red);
        Console.ReadKey();
    }
}

class ChangeAxisScale : GraphState
{
    private readonly Graph.Axis axis;

    public ChangeAxisScale(Graph graph, Graph.Axis axis) : base(graph)
    {
        this.axis = axis;
    }

    public override void Draw()
    {
        base.Draw();
        Graphics.PrintLine($"Enter the maximum {axis}, 'r' for the recommended or 'c' for the  
current", ConsoleColor.White);
    }

    public override void Update()
    {
        try
        {
            string input = Console.ReadLine();
            int maxSize = 0;

            if (input.ToLower().Trim() != "c")
            {
                if (input.ToLower().Trim() == "r")
                {
                    maxSize = axis == Graph.Axis.input ? graph.GreatestX() : graph.GreatestY();
                }
                else
                {

```

# Time complexity estimator

## Technical Solution

```
        if (!int.TryParse(input, out maxSize) || maxSize <= 0)
        {
            throw new InvalidInputException();
        }
    }

    if (axis == Graph.Axis.input)
    {
        graph.SetXAxis(maxSize);
    }
    else
    {
        graph.SetYAxis(maxSize);
    }
}

Continue();
}
catch (Exception ex)
{
    Console.WriteLine();
    Graphics.PrintLine(ex.Message, ConsoleColor.Red);
    Console.ReadKey();
}
}

private void Continue()
{
    if (axis == Graph.Axis.input)
    {
        StateManager.ChangeState(new ChangeAxisScale(graph, Graph.Axis.time));
    }
    else
    {
        StateManager.ChangeState(new DisplayResults(SavedResults.GetResults(),
true, graph));
    }
}

class ChangeAxisSize : GraphState
{
    private readonly Graph.Axis axis;

    public ChangeAxisSize(Graph graph, Graph.Axis axis) : base(graph)
    {
        this.axis = axis;
    }

    public override void Draw()
    {
        base.Draw();
        Graphics.PrintLine($"Enter the length of the {axis} axis in {(axis ==
Graph.Axis.input ? "columns" : "rows")}, 'r' for the recommended or 'c' for the current",
ConsoleColor.White);
    }

    public override void Update()
    {
        try
        {
```

# Time complexity estimator

## Technical Solution

```
string input = Console.ReadLine();
int length = 0;

if (input.ToLower().Trim() != "c")
{
    if (input.ToLower().Trim() == "r")
    {
        length = axis == Graph.Axis.input ? 50 : 20;
    }
    else
    {
        if (!int.TryParse(input, out length) || length < 15 || length >
Console.BufferWidth)
        {
            throw new InvalidInputException();
        }

        if (axis == Graph.Axis.input)
        {
            graph.SetXAxisLength(length);
        }
        else
        {
            graph.SetYAxisLength(length);
        }
    }

    Continue();
}
catch (Exception ex)
{
    Console.WriteLine();
    Graphics.PrintLine(ex.Message, ConsoleColor.Red);
    Console.ReadKey();
}

private void Continue()
{
    if (axis == Graph.Axis.input)
    {
        StateManager.ChangeState(new ChangeAxisSize(graph, Graph.Axis.time));
    }
    else
    {
        SavedResults.XAxisLength = graph.XAxisLength;
        SavedResults.YAxisLength = graph.YAxisLength;
        StateManager.ChangeState(new DisplayResults(SavedResults.GetResults(), true,
graph));
    }
}

class Extrapolate : GraphState
{
    private readonly Graph.Axis extrapolatedAxis;
    private readonly Graph.Axis resultAxis;
    private readonly int resultIndex;
```

# Time complexity estimator

## Technical Solution

```
public Extrapolate(Graph graph, Graph.Axis axis, int resultIndex) : base(graph)
{
    extrapolatedAxis = axis;
    resultAxis = extrapolatedAxis == Graph.Axis.input ? Graph.Axis.time : Graph.Axis.input;
    this.resultIndex = resultIndex;
}

public override void Draw()
{
    base.Draw();
    Graphics.Print($"Enter the extrapolated {extrapolatedAxis} from ", ConsoleColor.White);
    Graphics.Print($"'{graph.Results[resultIndex].complexity}':\n",
Graph.Colors[resultIndex]);
}

public override void Update()
{
    double num = Input.GetInput<double>();

    if (num != default && num > 0)
    {
        Console.WriteLine();
        Graphics.PrintLine($"The estimated {resultAxis} for the given {extrapolatedAxis}
{num}{{(extrapolatedAxis == Graph.Axis.input ? "" : "ms")}} is {Math.Max(Math.Round(extrapolatedAxis ==
Graph.Axis.input ? graph.Equations[resultIndex].SolveForY(num) :
graph.Equations[resultIndex].SolveForX(num)), 0)}{{(resultAxis == Graph.Axis.input ? "" :
"ms")}},\nusing {graph.Equations[resultIndex].Print()} as the generalised function",
ConsoleColor.Cyan);
        Console.ReadKey();

        StateManager.ChangeState(new DisplayResults(SavedResults.GetResults(), false));
    }
}

class GetAxis : GraphState
{
    private readonly int resultIndex = 0;

    public GetAxis(Graph graph, int resultIndex) : base(graph)
    {
        this.resultIndex = resultIndex;
    }

    public override void Draw()
    {
        ClearBottomOfScreen();
        Graphics.PrintLine("Enter 1 to extrapolate input, or 2 to extrapolated times",
ConsoleColor.White);
    }

    public override void Update()
    {
        int key = Input.KeyToInt(0, 3);

        if (key != -1)
        {
            StateManager.ChangeState(new Extrapolate(graph, key == 1 ? Graph.Axis.input :
Graph.Axis.time, resultIndex));
        }
    }
}

class ChangeAxis : GraphState
{
    private readonly Graph.Axis axis;
```

# Time complexity estimator

## Technical Solution

```
public ChangeAxis(Graph graph, Graph.Axis axis) : base(graph)
{
    this.axis = axis;
}

public override void Draw()
{
    base.Draw();
    Graphics.PrintLine($"Enter the maximum {axis}, 'm' for the maximum or 'c'
for the current", ConsoleColor.White);
}

public override void Update()
{
    try
    {
        string input = Console.ReadLine();
        int maxSize = 0;

        if (input.ToLower().Trim() != "c")
        {
            if (input.ToLower().Trim() == "m")
            {
                maxSize = axis == Graph.Axis.input ? graph.GreatestX() :
graph.GreatestY();
            }
            else
            {
                if (!int.TryParse(input, out maxSize) || maxSize <= 0)
                {
                    throw new InvalidInputException();
                }
            }

            if (axis == Graph.Axis.input)
            {
                graph.SetXAxis(maxSize);
            }
            else
            {
                graph.SetYAxis(maxSize);
            }
        }

        Continue();
    }
    catch (Exception ex)
    {
        Console.WriteLine();
        Graphics.PrintLine(ex.Message, ConsoleColor.Red);
        Console.ReadKey();
    }
}

private void Continue()
{
    if (axis == Graph.Axis.input)
    {
        StateManager.ChangeState(new ChangeAxis(graph, Graph.Axis.time));
    }
}
```

# Time complexity estimator

## Technical Solution

```
    }
    else
    {
        StateManager.ChangeState(new DisplayResults(SavedResults.GetResults(), true, graph));
    }
}

class DisplayResults : GraphState
{
    private readonly Menu menu;
    private readonly List<Result> results = new List<Result>();

    /// <param name="doFirstDraw"> Whether to draw the template for the graph or not </param>
    /// <param name="graph"> Optional paramter to pass an already instantiated graph to avoid a
new one being made </param>
    public DisplayResults(List<Result> results, bool doFirstDraw, Graph graph = null) : base(graph
?? new Graph(results))
    {
        this.results = results;
        SavedResults.SaveResults(results);
        menu = new Menu(
            "Please select your choice:",
            new MenuOption("Interrogate graph", Interrogate),
            new MenuOption("Extrapolate graph", ExtrapolateGraph),
            new MenuOption("Find intersection", FindIntersection),
            new MenuOption("Change the axis size or scale", ChangeAxis),
            new MenuOption("Add algorithms to this graph", AddAlgorithm),
            new MenuOption("Back to main menu", MainMenu)
        );

        if (doFirstDraw)
            FirstDraw();
    }

    /// <summary>
    /// Draws the necessary templates so that the entire screen does not need to be cleared
    /// </summary>
    public void FirstDraw()
    {
        DrawBanner();
        if (results.Count > 1)
        {
            Graphics.PrintLine($"Your algorithms are likely to be:", ConsoleColor.White);
            for (int i = 0; i < results.Count; i++)
            {
                Graphics.Print($"{results[i].complexity} - {results[i].bigO} {(i == results.Count -
1 ? "" : ",") } ", Graph.Colors[i]);
            }
        }
        else
        {
            Graphics.PrintLine($"Your algorithm is likely to be {results[0].complexity} -
{results[0].bigO}", ConsoleColor.White);
        }
        Console.WriteLine("\n");
        graph.Draw();
    }

    /// <exception cref="GraphLimitReachedException">
    /// Thrown when the user attempts to add more than 5 algorithms to the graph
    /// </exception>
    public void AddAlgorithm()
```

# Time complexity estimator

## Technical Solution

```
{
    if (results.Count >= 5)
    {
        throw new GraphLimitReachedException();
    }
    else
    {
        StateManager.ChangeState(new GetAlgorithm());
    }
}

public void Interrogate()
{
    StateManager.ChangeState(new Interrogate(graph));
}

public override void Draw()
{
    base.Draw();
    menu.Draw();
}

public override void Update()
{
    menu.Update();
    base.Update();
}

public void Quit()
{
    StateManager.ChangeState(new Quit());
}

public void ChangeAxis()
{
    ClearBottomOfScreen();
    Graphics.PrintLine("Press 1 to change the axis scale, or 2 to change the
axis size", ConsoleColor.Cyan);

    int key = Input.KeyToInt(1, 2);

    if (key == 1)
    {
        StateManager.ChangeState(new ChangeAxisScale(graph,
Graph.Axis.input));
    }
    else
    {
        StateManager.ChangeState(new ChangeAxisSize(graph, Graph.Axis.input));
    }
}

public void ExtrapolateGraph()
{
    if (graph.numOfResults == 1)
    {
        StateManager.ChangeState(new GetAxis(graph, 0));
    }
    else
    {
        StateManager.ChangeState(new GetAxis(graph, SelectResult()));
    }
}
```



# Time complexity estimator

## Technical Solution

```
    }
}

public void FindIntersection()
{
    if (graph.numOfResults == 1)
    {
        throw new NotEnoughResultsException();
    }
    else
    {
        ClearBottomOfScreen();
        int equation1Index = SelectResult();
        Equation equation1 = graph.Equations[equation1Index];
        ClearBottomOfScreen();
        Equation equation2 = graph.Equations[SelectResult(equation1Index)];
        StateManager.ChangeState(new FindIntersection(graph, equation1,
equation2));
    }
}

public void MainMenu()
{
    SavedResults.RemoveResults();
    StateManager.ChangeState(new Welcome());
}
}

class FindIntersection : GraphState
{
    Graph.Vector intersect;

    public FindIntersection(Graph graph, Equation equation1, Equation equation2) :
base(graph)
    {
        intersect = graph.FindIntersection(equation1, equation2);
    }

    public override void Draw()
    {
        base.Draw();
        if (intersect.X < 0 || intersect.Y < 0)
        {
            Graphics.Print("No intersections found", ConsoleColor.Red);
        }
        else
        {
            Graphics.PrintLine($"These graphs intersect at:", ConsoleColor.Cyan);
            Graphics.PrintLine($"Input: {intersect.X}\nTime: {intersect.Y}",
ConsoleColor.White);
        }
    }

    public override void Update()
    {
        base.Update();
        Console.ReadKey();
        StateManager.ReturnToLastState();
    }
}

class Interrogate : GraphState
```

# Time complexity estimator

## Technical Solution

```
{  
  
    int resultIndex;  
    int nodeIndex;  
  
    public Interrogate(Graph graph) : base(graph)  
    {  
        resultIndex = graph.numOfResults - 1;  
        nodeIndex = 0;  
        graph.ChangeSelectedNode(resultIndex, nodeIndex);  
        ClearBottomOfScreen();  
        ShowNodeInfoTemplate();  
    }  
  
    public override void Draw()  
    {  
        graph.RefreshInterrogateDraw();  
        Graphics.PrintLine("Use left and right to change node,\nup and down to change  
algorithm\nor 'q' to return to the menu", ConsoleColor.Cyan);  
    }  
  
    public override void Update()  
    {  
        ConsoleKey key = Console.ReadKey(true).Key;  
  
        switch (key)  
        {  
            case ConsoleKey.LeftArrow:  
                nodeIndex = ((nodeIndex - 1) % Graph.NumOfNodes < 0) ? Graph.NumOfNodes  
- 1 : ((nodeIndex - 1) % Graph.NumOfNodes);  
  
                if (!graph.IsNodeVisible(resultIndex, nodeIndex))  
                {  
                    nodeIndex = Graph.NumOfNodes;  
                    do  
                    {  
                        nodeIndex--;  
                        nodeIndex = Math.Max(0, nodeIndex);  
                    } while (!graph.IsNodeVisible(resultIndex, nodeIndex) && nodeIndex  
> 0);  
                }  
  
                graph.ChangeSelectedNode(resultIndex, nodeIndex);  
                break;  
            case ConsoleKey.RightArrow:  
                nodeIndex = (nodeIndex + 1) % Graph.NumOfNodes;  
  
                if (!graph.IsNodeVisible(resultIndex, nodeIndex))  
                {  
                    nodeIndex = 0;  
                }  
  
                graph.ChangeSelectedNode(resultIndex, nodeIndex);  
                break;  
            case ConsoleKey.DownArrow:  
                graph.ChangeSelectedNode(-1);  
                resultIndex = ((resultIndex - 1) % graph.numOfResults < 0) ?  
graph.numOfResults - 1 : ((resultIndex - 1) % graph.numOfResults);  
  
                if (!graph.IsNodeVisible(resultIndex, nodeIndex))  
                {  
                    nodeIndex = Graph.NumOfNodes;  
                    do
```

# Time complexity estimator

## Technical Solution

```
        {
            nodeIndex--;
            nodeIndex = Math.Max(0, nodeIndex);
        } while (!graph.IsNodeVisible(resultIndex, nodeIndex) && nodeIndex > 0);
    }

    graph.ChangeSelectedNode(resultIndex, nodeIndex);
    break;
case ConsoleKey.UpArrow:
    graph.ChangeSelectedNode(-1);
    resultIndex = ((resultIndex + 1) % graph.numOfResults < 0) ? graph.numOfResults + 1 :
((resultIndex + 1) % graph.numOfResults);

    if (!graph.IsNodeVisible(resultIndex, nodeIndex))
    {
        nodeIndex = Graph.NumOfNodes;
        do
        {
            nodeIndex--;
            nodeIndex = Math.Max(0, nodeIndex);
        } while (!graph.IsNodeVisible(resultIndex, nodeIndex) && nodeIndex > 0);
    }

    graph.ChangeSelectedNode(resultIndex, nodeIndex);
    break;
case ConsoleKey.Q:
    graph.ChangeSelectedNode(-1);
    ClearBottomOfScreen();
    StateManager.ReturnToLastState();
    break;
    }
}

public void ShowNodeInfoTemplate()
{
    Graphics.PrintLine("Selected Node:", ConsoleColor.Red);
    Graphics.PrintLine("Time:", ConsoleColor.Cyan);
    Graphics.PrintLine("Input Size:", ConsoleColor.Cyan);
    Console.WriteLine();
}

static class SavedResults
{
    private static List<Result> results;

    public static int XAxisLength { get; set; } = Graph.DefaultXAxisLength;
    public static int YAxisLength { get; set; } = Graph.DefaultYAxisLength;

    public static void SaveResults(List<Result> resultsToSave)
    {
        results = resultsToSave;
    }

    public static void SaveGraphAxisLengths(int xAxisLength, int yAxisLength)
    {
        XAxisLength = xAxisLength;
        YAxisLength = yAxisLength;
    }
}
```

# Time complexity estimator

## Technical Solution

```
public static void RemoveResults()
{
    results = null;
    XAxisLength = Graph.DefaultXAxisLength;
    YAxisLength = Graph.DefaultYAxisLength;
}

public static List<Result> GetResults()
{
    return results;
}

public static void AddResult(Result result)
{
    results.Add(result);
}

public static bool AreResultsSaved()
{
    return !(results == null);
}
}

class SelectResult : GraphState
{
    private List<Result> results = new List<Result>();
    private int removeIndex;

    public int ResultIndex { get; private set; }
    public bool HasFinished { get; private set; } = false;

    /// <param name="removeThisIndex">
    /// Optional parameter to remove a result from the options to choose from
    /// </param>
    public SelectResult(Graph graph, int removeThisIndex = -1) : base(graph)
    {
        for (int i = 0; i < graph.Results.Count; i++)
        {
            if (i != removeThisIndex)
                results.Add(graph.Results[i]);
        }
        removeIndex = removeThisIndex;
        ClearBottomOfScreen();
        PrintResults();
        Console.WriteLine();
        Graphics.PrintLine("Use up and down to select a result,\nusing enter to select it", ConsoleColor.Cyan);
    }

    public override void Draw()
    {
        Graphics.Print(" ", ConsoleColor.Red);
        Console.SetCursorPosition(results[ResultIndex].complexity.ToString().Length + 1, graph.YAxisLength + 10 + ResultIndex);
        Graphics.Print("*", ConsoleColor.Red);
    }

    public override void Update()
    {
        Console.SetCursorPosition(results[ResultIndex].complexity.ToString().Length + 1, graph.YAxisLength + 10 + ResultIndex);
    }
}
```

# Time complexity estimator

## Technical Solution

```
        ConsoleKey key = Console.ReadKey(true).Key;

        switch (key)
        {
            case ConsoleKey.UpArrow:
                ResultIndex = ((ResultIndex - 1) % results.Count < 0) ? results.Count - 1 : ((ResultIndex - 1) % results.Count);
                break;
            case ConsoleKey.DownArrow:
                ResultIndex = ((ResultIndex + 1) % results.Count < 0) ? results.Count + 1 : ((ResultIndex + 1) % results.Count);
                break;
            case ConsoleKey.Enter:
                ClearBottomOfScreen();
                ResultIndex = graph.Results.FindIndex(0, x =>
                    x.Equals(results[ResultIndex]));
                HasFinished = true;
                break;
        }
    }

    public void PrintResults()
    {
        for (int i = 0; i < graph.Equations.Count; i++)
        {
            if (i != removeIndex)
                Graphics.PrintLine($"{graph.Results[i].complexity}", Graph.Colors[i]);
        }
    }
}

abstract class Equation
{
    protected double[] coefficients;
    protected double[] derivativeCoefficients;

    public Equation(Result result)
    {
        coefficients = CalculateCoefficient(result);
        derivativeCoefficients = CalculateDerivativeCoefficient();
    }

    public virtual double SolveForY(double x)
    {
        throw new NotImplementedException();
    }

    public virtual double SolveForX(double y)
    {
        throw new NotImplementedException();
    }

    public virtual double SolveDerivative(double x)
    {
        throw new NotImplementedException();
    }

    public virtual string Print()
    {
        throw new NotImplementedException();
    }
}
```

# Time complexity estimator

## Technical Solution

```
    public virtual double[] CalculateCoefficient(Result result)
    {
        throw new NotImplementedException();
    }

    public virtual double[] CalculateDerivativeCoefficient()
    {
        throw new NotImplementedException();
    }
}

class ConstantEquation : Equation
{
    // y = [0]

    public ConstantEquation(Result result) : base(result)
    {
    }

    public override double SolveForY(double x)
    {
        return coefficients[0];
    }

    public override double SolveForX(double y)
    {
        return double.NaN;
    }

    public override double SolveDerivative(double x)
    {
        return derivativeCoefficients[0];
    }

    public override string Print()
    {
        return $"y = {Math.Round(coefficients[0], 5)}";
    }

    public override double[] CalculateCoefficient(Result result)
    {
        return new double[1] { MathHelper.Mean(result.times) };
    }

    public override double[] CalculateDerivativeCoefficient()
    {
        return new double[1] { 0 };
    }
}

class LinearEquation : Equation
{
    // y = [0]x + [1]

    public LinearEquation(Result result) : base(result)
    {
    }
}
```

# Time complexity estimator

## Technical Solution

```
}

public override double SolveForY(double x)
{
    return coefficients[0] * x + coefficients[1];
}

public override double SolveForX(double y)
{
    return (y - coefficients[1]) / coefficients[0];
}

public override double SolveDerivative(double x)
{
    return derivativeCoefficients[0];
}

public override string Print()
{
    return $"y = {coefficients[0]:E1}x + {Math.Round(coefficients[1], 2)}";
}

public override double[] CalculateCoefficient(Result result)
{
    double c = result.times[0];

    double totalGradient = 0;
    for (int i = 1; i < result.times.Length; i++)
    {
        totalGradient += (double)(result.times[i] - result.times[i - 1]) /
(result.inputs[i] - result.inputs[i - 1]);
    }
    double gradient = totalGradient / (result.times.Length - 1);

    return new double[2] { gradient, c };
}

public override double[] CalculateDerivativeCoefficient()
{
    return new double[1] { coefficients[0] };
}
}

class QuadraticEquation : Equation
{
    // y = [0]x² + [1]
    // There is no x term as the graph will always have its turning point at x = 0

    public QuadraticEquation(Result result) : base(result)
    {
    }

    public override double SolveForY(double x)
    {
        return coefficients[0] * Math.Pow(x, 2) + coefficients[1];
    }
}
```

# Time complexity estimator

## Technical Solution

```
public override double SolveForX(double y)
{
    return Math.Sqrt((y - coefficients[1]) / coefficients[0]);
}

public override double SolveDerivative(double x)
{
    return coefficients[0] * x;
}

public override string Print()
{
    return $"y = {coefficients[0]:E2}x2 + {coefficients[1]}";
}

public override double[] CalculateCoefficient(Result result)
{
    double c = result.times[0];

    double totalA = 0;
    for (int i = 1; i < result.times.Length; i++)
    {
        totalA += (double)(result.times[i] - c) / Math.Pow(result.inputs[i],
2);
    }
    double a = totalA / (result.times.Length - 1);

    return new double[2] { a, c };
}

public override double[] CalculateDerivativeCoefficient()
{
    return new double[1] { 2 * coefficients[0] };
}
}

class CubicEquation : Equation
{
    // y = [0]x3 + [1]
    // There is no x2 or x term as the graph will always have its turning point at
x = 0

    public CubicEquation(Result result) : base(result)
    {
    }

    public override double SolveForY(double x)
    {
        return coefficients[0] * Math.Pow(x, 3) + coefficients[1];
    }

    public override double SolveForX(double y)
    {
        return Math.Cbrt((y - coefficients[1]) / coefficients[0]);
    }

    public override double SolveDerivative(double x)
    {
        return derivativeCoefficients[0] * Math.Pow(x, 2);
    }
}
```



# Time complexity estimator

## Technical Solution

```
    }

    public override string Print()
    {
        return $"y = {coefficients[0]:E2}x3 + {coefficients[1]}";
    }

    public override double[] CalculateCoefficient(Result result)
    {
        double c = result.times[0];

        // Uses later coordinates for accuracy

        double totalA = 0;
        for (int i = 3; i < result.times.Length; i++)
        {
            totalA += (double)(result.times[i] - c) / Math.Pow(result.inputs[i], 3);
        }
        double a = totalA / (result.times.Length - 3);

        return new double[2] { a, c };
    }

    public override double[] CalculateDerivativeCoefficient()
    {
        return new double[1] { 3 * coefficients[0] };
    }
}

class LogarithmicEquation : Equation
{
    // y = [0] log(x) + [1]

    public LogarithmicEquation(Result result) : base(result)
    {
    }

    public override double SolveForY(double x)
    {
        return coefficients[0] * Math.Log2(x) + coefficients[1];
    }

    public override double SolveForX(double y)
    {
        return Math.Pow(2, (y - coefficients[1]) / coefficients[0]);
    }

    public override double SolveDerivative(double x)
    {
        return derivativeCoefficients[0] / x;
    }

    public override string Print()
    {
        return $"y = {Math.Round(coefficients[0], 2)}log(x) + {Math.Round(coefficients[1], 2)}";
    }
}
```

# Time complexity estimator

## Technical Solution

```
public override double[] CalculateCoefficient(Result result)
{
    double[] logInputs = new double[result.inputs.Length];

    for (int i = 1; i < result.inputs.Length; i++)
    {
        logInputs[i] = Math.Log2(result.inputs[i]);
    }

    double gradient = 0;
    for (int i = 2; i < result.times.Length; i++)
    {
        gradient += (result.times[i] - result.times[i - 1]) / (logInputs[i] -
logInputs[i - 1]);
    }
    gradient /= result.times.Length - 2;

    double c = 0;
    for (int i = 1; i < result.times.Length; i++)
    {
        c += result.times[i] - (gradient * logInputs[i]);
    }
    c /= result.times.Length - 1;

    return new double[2] { gradient, c };
}

public override double[] CalculateDerivativeCoefficient()
{
    return new double[1] { coefficients[0] / Math.Log(2) };
}
}

class ExponentialEquation : Equation
{
    // y = [0] * 2^x + [1]

    public ExponentialEquation(Result result) : base(result)
    {
    }

    public override double SolveForY(double x)
    {
        return coefficients[0] * Math.Pow(2, x) + coefficients[1];
    }

    public override double SolveForX(double y)
    {
        return Math.Log2((y - coefficients[1]) / coefficients[0]);
    }

    public override double SolveDerivative(double x)
    {
        return Math.Pow(2, x) * derivativeCoefficients[0];
    }

    public override string Print()
    {
    }
}
```

# Time complexity estimator

## Technical Solution

```
        return $"y = {coefficients[0]:E2} * 2^x + {coefficients[1]}";
    }

    public override double[] CalculateCoefficient(Result result)
    {
        double c = result.times[0];

        double a = (double)(result.times[^1] - c) / Math.Pow(2, result.inputs[^1]);

        return new double[2] { a, c };
    }

    public override double[] CalculateDerivativeCoefficient()
    {
        return new double[1] { Math.Log(2) * coefficients[0] };
    }
}

class Graph : IDrawable
{
    public const int NumOfNodes = 10;
    public const int DefaultXAxisLength = 50;
    public const int DefaultYAxisLength = 20;

    public static readonly ConsoleColor[] Colors = new ConsoleColor[5] { ConsoleColor.White,
        ConsoleColor.Magenta, ConsoleColor.Green, ConsoleColor.Yellow, ConsoleColor.Gray };

    private readonly List<Node[]> nodes = new List<Node[]>();
    private int[] xAxis;
    private int[] yAxis;
    private Coordinate[,] points;
    private float xInterval;
    private int yInterval;
    private int xAxisScaleFactor;
    public int numOfResults;

    public int XAxisLength { get; private set; }
    public int YAxisLength { get; private set; }
    public int NodeIndex { private get; set; }
    public int ResultIndex { private get; set; }
    public List<Equation> Equations { get; private set; } = new List<Equation>();
    public List<Result> Results { get; private set; } = new List<Result>();

    public Graph(List<Result> results, int xAxisLength = DefaultXAxisLength, int yAxisLength =
        DefaultYAxisLength)
    {
        Results = results;
        numOfResults = results.Count;
        XAxisLength = xAxisLength;
        YAxisLength = yAxisLength;
        SetNodes(results);
        SetXAxis(GreatestX());
        SetYAxis(GreatestY());
        AddEquation(results);
        SetPoints();
    }

    public void Draw()
    {

```

# Time complexity estimator

## Technical Solution

```
Graphics.PrintLine($"Time (ms)", ConsoleColor.Cyan);

for (int y = 0; y < YAxisLength; y++)
{
    Graphics.Print($"{yAxis[^y + 1].ToString().PadRight(yAxis[^1].ToString().Length)}|",
        ConsoleColor.DarkCyan);

    for (int x = 0; x < XAxisLength; x++)
    {
        points[y, x].Draw();
    }

    Console.WriteLine();
}

Graphics.Print($"{new string(' ', yAxis[^1].ToString().Length)}L{new string('-',
XAxisLength)}", ConsoleColor.DarkCyan);
Console.WriteLine();
Console.Write(new string(' ', yAxis[^1].ToString().Length + 1));

for (int x = 0; x < XAxisLength; x += 3)
{
    string num = xAxis[x].ToString().PadLeft(Math.Max(xAxis[^1].ToString().Length, 2),
'0');
    Graphics.Print(num.ToCharArray()[0].ToString() + num.ToCharArray()[1].ToString() + "
", ConsoleColor.DarkCyan);
}

Console.WriteLine();
Graphics.PrintLine($"{new string(' ', yAxis[^1].ToString().Length + 1 + XAxisLength / 2 -
(8 + Convert.ToDouble(xAxisScaleFactor).ToString("E1").Length) / 2)}Input
(x{Convert.ToDouble(xAxisScaleFactor):E1})", ConsoleColor.Cyan);
Console.WriteLine("\n");
}

private void RefreshNodesOfCurrentResult()
{
    foreach (Node node in nodes[ResultIndex])
    {
        if (node.isVisible)
        {
            Console.SetCursorPosition(yAxis[^1].ToString().Length + 1 + (int)(node.position.X
/ xInterval), 5 + YAxisLength - (int)(node.position.Y / yInterval));
            node.Draw();
        }
    }
}

public void RefreshInterrogateDraw()
{
    RefreshNodesOfCurrentResult();
    Console.SetCursorPosition(6, 11 + YAxisLength);
    nodes[ResultIndex][NodeIndex].PrintTime();
    Console.SetCursorPosition(12, 12 + YAxisLength);
    nodes[ResultIndex][NodeIndex].PrintInput();
    Console.SetCursorPosition(0, 13 + YAxisLength);
    Graphics.Print($"From algorithm: ", ConsoleColor.White);
    Graphics.Print($"{Results[ResultIndex].complexity}'{new string(' ', Console.WindowWidth -
Results[ResultIndex].complexity.ToString().Length - 18)}", Colors[ResultIndex]);
    Console.SetCursorPosition(0, 15 + YAxisLength);
}

/// <summary>
/// Changes the currently selected node
/// </summary>
/// <param name="resultIndex"> Pass '-1' to remove the selected node </param>
public void ChangeSelectedNode(int resultIndex, int nodeIndex = 0)
```

# Time complexity estimator

## Technical Solution

```
{
    nodes[ResultIndex][NodeIndex].IsSelected = false;
    if (resultIndex != -1)
    {
        ResultIndex = resultIndex;
        NodeIndex = nodeIndex;
        nodes[ResultIndex][NodeIndex].IsSelected = true;
    }

    RefreshNodesOfCurrentResult();
}

public bool IsNodeVisible(int resultIndex, int nodeIndex)
{
    return nodes[resultIndex][nodeIndex].isVisible;
}

private void SetNodes(List<Result> results)
{
    int colorCount = 0;

    foreach (Result result in results)
    {
        nodes.Add(new Node[result.times.Length]);
        for (int i = 0; i < result.times.Length; i++)
        {
            nodes[^1][i] = new Node(new Vector(result.inputs[i],
result.times[i]), Colors[colorCount % 5]);
        }
        colorCount++;
    }
}

public void SetXAxis(int end)
{
    xAxis = new int[XAxisLength];

    xInterval = (float)end / (XAxisLength - 1);
    float totalX = 0;

    for (int i = 0; i < XAxisLength; i++)
    {
        xAxis[i] = (int)totalX;
        totalX += xInterval;
    }

    xAxisScaleFactor = (int)Math.Pow(10, Math.Max(0,
xAxis[^1].ToString().Length - 2));

    SetPoints();
}

public void SetYAxis(int end)
{
    yAxis = new int[YAxisLength];

    yInterval = (int)Math.Ceiling((float)end / (YAxisLength - 1));
    float totalY = 0;

    for (int i = 0; i < YAxisLength; i++)
```

# Time complexity estimator

## Technical Solution

```
{
    yAxis[i] = (int)totalY;
    totalY += yInterval;
}

SetPoints();
}

public void SetXAxisLength(int XAxisLength)
{
    this.XAxisLength = XAxisLength;
    SetXAxis(xAxis[^1]);
    SetPoints();
}

public void SetYAxisLength(int YAxisLength)
{
    this.YAxisLength = YAxisLength;
    SetYAxis(yAxis[^1]);
    SetPoints();
}

private void SetPoints()
{
    int i = 0;
    int xIndex;
    int yIndex;

    points = new Coordinate[YAxisLength, XAxisLength];

    for (int y = 1; y <= YAxisLength; y++)
    {
        for (int x = 0; x < XAxisLength; x++)
        {
            points[YAxisLength - y, x] = new Coordinate(new Vector((int)(x *
xInterval), (int)(y * yInterval)), " ", ConsoleColor.Black);
        }
    }

    foreach (Equation equation in Equations)
    {
        xIndex = 0;
        yIndex = 0;
        List<Node> nodesToDraw = new List<Node>();

        foreach (Node node in nodes[i])
        {
            node.isVisible = false;
            nodesToDraw.Add(node);
        }

        for (float y = yAxis[0]; y <= yAxis[^1]; y += yInterval)
        {
            for (float x = xAxis[0]; x <= xAxis[^1]; x += xInterval)
            {
                string symbol = " ";
                double solveForY = equation.SolveForY(x);
                double solveForYDecimal = (solveForY % yInterval) / yInterval;
                double solveForX = equation.SolveForX(y);
                double solveForXDecimal = (solveForX % xInterval) / xInterval;
```

# Time complexity estimator

## Technical Solution

```
if ((int)(solveForY / yInterval) == (int)(y / yInterval))
{
    if (solveForYDecimal < 0.33)
    {
        symbol = ".";
    }
    else if (solveForYDecimal > 0.66)
    {
        symbol = "'";
    }
    else
    {
        symbol = ".";
    }

    points[YAxisLength - 1 - yIndex, xIndex] = new Coordinate(new
Vector((int)x, (int)y), symbol, Colors[i]);
}
else if (Math.Ceiling(solveForX / xInterval) == xIndex)
{
    symbol = ".";

    points[YAxisLength - 1 - yIndex, xIndex] = new Coordinate(new
Vector((int)x, (int)y), symbol, Colors[i]);
}

foreach (Node node in nodesToDraw)
{
    if ((int)(node.position.X / xInterval) == xIndex &&
node.position.Y / yInterval == yIndex)
    {
        node.isVisible = true;
        nodesToDraw.Remove(node);
        points[YAxisLength - 1 - yIndex, xIndex] = node;
        break;
    }
}

xIndex++;
}

xIndex = 0;
yIndex++;
}

i++;
}

public void AddEquation(List<Result> results)
{
    foreach (Result result in results)
    {
        switch (result.complexity)
        {
            case Complexity.Constant:
                Equations.Add(new ConstantEquation(result));
                break;
            case Complexity.Logarithmic:
                Equations.Add(new LogarithmicEquation(result));
                break;
            case Complexity.Linear:
                Equations.Add(new LinearEquation(result));
                break;
        }
    }
}
```

# Time complexity estimator

## Technical Solution

```
        Equations.Add(new LinearEquation(result));
        break;
    case Complexity.Quadratic:
        Equations.Add(new QuadraticEquation(result));
        break;
    case Complexity.Cubic:
        Equations.Add(new CubicEquation(result));
        break;
    case Complexity.Exponential:
        Equations.Add(new ExponentialEquation(result));
        break;
    }
}

public Vector FindIntersection(Equation equation1, Equation equation2)
{
    double xIntersect = 1;

    for (int i = 0; i < 100; i++)
    {
        xIntersect = xIntersect - (equation1.SolveForY(xIntersect) -
equation2.SolveForY(xIntersect)) / (equation1.SolveDerivative(xIntersect) -
equation2.SolveDerivative(xIntersect));
    }

    return new Vector((int)xIntersect, (int)equation1.SolveForY(xIntersect));
}

public int GreatestX()
{
    int greatestX = nodes[0][0].position.X;

    foreach (Node[] nodeArray in nodes)
    {
        for (int i = 1; i < nodeArray.Length; i++)
        {
            if (nodeArray[i].position.X > greatestX)
            {
                greatestX = nodeArray[i].position.X;
            }
        }
    }

    return greatestX;
}

public int GreatestY()
{
    int greatestY = nodes[0][0].position.Y;

    foreach (Node[] nodeArray in nodes)
    {
        for (int i = 1; i < nodeArray.Length; i++)
        {
            if (nodeArray[i].position.Y > greatestY)
            {
                greatestY = nodeArray[i].position.Y;
            }
        }
    }
}
```



# Time complexity estimator

## Technical Solution

```
    }

    return greatestY;
}

public struct Vector
{
    public int X { get; set; }
    public int Y { get; set; }

    public Vector(int x, int y)
    {
        X = x;
        Y = y;
    }
}

private class Coordinate : IDrawable
{
    public readonly Vector position;
    public readonly ConsoleColor color;
    protected readonly string symbol;

    public Coordinate(Vector position, string symbol, ConsoleColor color)
    {
        this.position = position;
        this.symbol = symbol;
        this.color = color;
    }

    public virtual void Draw()
    {
        Graphics.Print(symbol, color);
    }
}

private class Node : Coordinate
{
    public bool IsSelected { private get; set; }
    public bool isVisible;

    public Node(Vector position, ConsoleColor color) : base(position, "X",
color)
    {
    }

    public override void Draw()
    {
        Graphics.Print(symbol, IsSelected ? ConsoleColor.Red : color);
    }

    public void PrintInput()
    {
        Graphics.Print($"{position.X}{new string(' ', Console.WindowWidth -
position.X.ToString().Length - 12)}", ConsoleColor.White);
    }

    public void PrintTime()
    {

```

# Time complexity estimator

## Technical Solution

```
        Graphics.Print($"{position.Y}ms{new string(' ', Console.WindowWidth -
position.Y.ToString().Length - 8)}", ConsoleColor.White);
    }

    }

    public enum Axis
    {
        input,
        time
    }
}

static class Graphics
{
    public static void PrintLine(string text, ConsoleColor color)
    {
        Console.ForegroundColor = color;
        Console.WriteLine(text);
        Console.ResetColor();
    }

    public static void Print(string text, ConsoleColor color)
    {
        Console.ForegroundColor = color;
        Console.Write(text);
        Console.ResetColor();
    }
}

class Menu : IDrawable, IUpdateable
{
    private readonly string banner;
    private readonly MenuOption[] actions;

    public Menu(string banner, params MenuOption[] actions)
    {
        this.banner = banner;
        this.actions = actions;
    }

    public void Draw()
    {
        Graphics.PrintLine(banner, ConsoleColor.Cyan);

        for (int i = 0; i < actions.Length; i++)
        {
            Graphics.PrintLine($"{i + 1} - {actions[i].description}",
ConsoleColor.White);
        }
    }

    public void Update()
    {
        int key = Input.KeyToInt(0, actions.Length);

        if (key != -1)
        {
            try
            {
                actions[key - 1].action.Invoke();
            }
            catch { }
        }
    }
}
```

# Time complexity estimator

## Technical Solution

```
        }
        catch (Exception ex)
        {
            Console.WriteLine();
            Graphics.PrintLine(ex.Message, ConsoleColor.Red);
            Console.ReadKey();
        }
    }
}

struct MenuOption
{
    public readonly string description;
    public readonly Action action;

    public MenuOption(string description, Action action)
    {
        this.description = description;
        this.action = action;
    }
}

static class Input
{
    /// <summary>
    /// Will take user input and attempt to cast it to T
    /// </summary>
    /// <returns>
    /// The inputted data or the default value of T if the cast was invalid
    /// </returns>
    public static T GetInput<T>() where T : struct
    {
        T result;
        try
        {
            result = (T)Convert.ChangeType(Console.ReadLine(), typeof(T));
        }
        catch (Exception)
        {
            Console.WriteLine();
            Graphics.PrintLine(new InvalidInputException().Message,
ConsoleColor.Red);
            Console.ReadKey();
            result = default;
        }

        return result;
    }

    /// <summary>
    /// Will take a user key press and attempt to cast it to an int
    /// </summary>
    /// <returns>
    /// The int or -1 if it was unsuccessful
    /// </returns>
    public static int KeyToInt(int min = 0, int max = int.MaxValue)
    {
        int result;
```

# Time complexity estimator

## *Technical Solution*

```
        try
        {
            result = int.Parse(Console.ReadKey().KeyChar.ToString());
            if (result < min || result > max)
            {
                throw new Exception();
            }
        }
        catch (Exception)
        {
            Console.WriteLine();
            Graphics.PrintLine(new InvalidInputException().Message,
ConsoleColor.Red);
            Console.ReadKey();
            result = -1;
        }

        return result;
    }
}
```

# Time complexity estimator

## Evaluation

### Evaluation of objectives:

1. The main menu worked as expected and correctly sanitised input, with a suitable message being printed out should the input not be in the correct format. The information printed about the program was informative but could have been improved. A full user manual could have been included under this option, as some options can be a little confusing if you do not have prior knowledge. An example of this would be the option to extrapolate the graph, which is not intuitive for a user who is not familiar with manipulation of graphs. The quit option correctly asks the user to re-affirm their input, an option that I believe should be included in more programs, as an accidental input can happen.

I consider this objective a success.

2. The inputting of the algorithm works irrespective of file extension, which has both positives and negatives. If the user entered 'constant' it would correctly run the 'constant.cs' C# file, the same would happen if the user entered 'constant.cs'. However, if the user entered 'constant.png' it would also run 'constant.cs', due to the program simply truncating the input before the dot. In hindsight, this is not the best way of sanitising the input, as a user might have incorrectly inputted a file of the wrong file extension but would not have been made aware of this. An error saying 'This file is of the wrong file extension' would have been more informative and user-friendly. Another potential issue is that the user can input the word 'quit' to return back to the menu but should 'quit' be the name of their program, then there is no way to run it unless they add the file extension. A better word to return to the menu could have been '/quit' as a file is unlikely to contain a slash. The program does identify if the file exists or not, and outputs a clear and informative error message.

I consider this object somewhat of a success.

3. The program is able to build and run the inputted C# files. The building of the program works, and the program will output any errors in the code back to the user. The running of the program is mostly a success, but an option to be able to abort the running of the algorithm would have been beneficial, as sometimes the user's program will hang, or it simply takes too long to execute. Currently the only way to fix this is to close the program and open it again, which is obviously not the best method. The multi-threading of the program allows a progress bar to be shown whilst the user is waiting. This feature was much more of a success than I anticipated, as the progress bar is much more accurate than I previously thought it would be.

I consider this objective a success.

4. The calculating of the time complexity worked far better than I imagined. The dynamic data calibration worked in most scenarios, but not all. Problems occurred when the program for some reason takes longer for a smaller amount of data, in which case the running of the program takes a long time. I cannot think of a way to fix this without calibrating every input, which would probably take longer than just running the program without calibrating the inputs at all. All time complexities were correctly identified at least at the 80% mark for all simple test programs, which my end-user stated was the minimum acceptable rate. I also tested a Bubble Sort as this is the sort of algorithm that I expect will be inputted to my program, and it correctly identified it as quadratic. From this test, I can see that this program works for not just small test programs but

# Time complexity estimator

## Evaluation

also for useful algorithms. This means that my program has fulfilled its objective to be used by programmers to test their algorithms.

I consider this objective a success.

5. The displaying of the results in a graph looks brilliant. The results are displayed clearly and effectively. The graph is intuitive to read thanks to the scaled axis, and highly customisable in both size and scale. The plots are clear and distinct due to the colour scheme, and the curve of best fit looks smooth and without gaps. There is a wealth of options available to the user to get data from their algorithms, many more than what I had originally planned. The interrogation option works in most cases but a few times when the axes are scaled just so, the selected node cannot be changed. I haven't been able to reliably reproduce this error, but it is still a nuisance nonetheless. The extrapolation options work brilliantly and can be applied to any of the plots. This definitely provides a practical solution to the problem of scaling up algorithms, which my end user said was important: *"For some programs you need to have an idea of how long they will take to run once you scale up the problem you are trying to solve"*. If the user enters a number too large for an integer data type when extrapolating a graph, the program will say 'Invalid Input' to the user. This is not clear to the user, as the input seems valid unless you have a knowledge of the language. A better error would have spelt out to the user that this number is too large. The intersection calculator works far better than I originally thought, and I haven't been able to get an incorrect result from it at all. If there are multiple intersection points, the program will simply give the one closest to the origin. Perhaps a message warning the user that there may be more intersections may have cleared up any confusion. A downside that I noticed after completing the code was that there is no indication as to which plots were for which files. I should have included a key for which plot was which filename, as this will make comparing them easier if you have 3 or more graphs. The last problem I found was with finding the curve of best fit for an exponential function. I originally tried an algorithm similar to the polynomial, that is, rearranging for a coefficient and then finding the mean of the results. This made a curve that did not match the curve generated at all. In the end I only solved for the coefficient from the last node, so that the curve will always go through this node. This looks better, but still not perfect. It also means that sometimes it does not fit the data at all. I'm not sure how I would fix this, perhaps a different algorithm is needed that does not deal with solving coefficients.

I consider this objective a great success.

# Time complexity estimator

## Evaluation

### Feedback from my end-user

After looking at my solution, my end-user provided the following feedback:

*“Very interesting to see the visualisation of the different algorithms and to see the constant, linear, quadratics. You provided a good selection of options. I particularly liked being able to extrapolate to see how long a larger run would take. It was also a nice touch to be able to compare with a constant algorithm to see the cut off point for which would be more efficient. It would be useful to have a name for the algorithm that is being tested and perhaps to see the extrapolated point on the graph.”*

I am definitely in agreement with this feedback. The criticism at the end is entirely valid, the first of which I already identified, but the second I overlooked. Once the user had extrapolated the graph, a point could have been plotted on the graph to provide a more answer. This technique could also be applied to the intersection point, so that the program will specifically highlight where it estimated the intersection to be.