ENGF0002

Scenario 1

Design Report

SACAT

Sorting Algorithm Complexity Analysis Tool

Team 10

Authors & Student Numbers:

Andrzej Szablewski	20016059
Tim Widmayer	20006018
Szymon Duchniewicz	20019971
Bartosz Grabek	20016589

University College London

2020/2021

Contents

1 Introduction **2**
1.1 Motivation . . . 2
1.2 Aim . . . 2

2 Application Design **2**
2.1 Program structure . . . 2
2.2 Resources . . . 2
2.3 GUI . . . 3

3 Methodology of Analysis **4**
3.1 Time complexity based on time . . . 4
3.2 Time complexity based on operation count . . . 4
3.3 Space complexity . . . 4

4 Core Functionality **5**
4.1 User input . . . 5
4.2 Preliminary Testing . . . 5
4.3 Data gathering . . . 6
4.3.1 Time limits . . . 6
4.3.2 Groups of tests . . . 6
4.4 Data analysis . . . 7
4.5 Example Analyses . . . 7
4.5.1 Mergesort . . . 7
4.5.2 Heapsort . . . 8
4.5.3 Bubblesort . . . 8

5 Possible Future Improvements **9**

List of Figures

1 GUI Overview . . . 3
2 Core general structure . . . 5
3 Mergesort data . . . 7
4 Heapsort data . . . 8
5 Bubblesort data . . . 8

1 Introduction

1.1 Motivation

Understanding the complexity of algorithms is a challenging task, in particular for novice programmers. As algorithms get more sophisticated, it becomes more difficult to analyse them and more important to test them in various circumstances. Comparing different algorithms without proper analytic tools can be tedious and time-consuming. There is a variety of algorithm implementations for any given problem, each with its own strengths and limitations. Therefore, a detailed and accurate comparison between different implementations is crucial.

The first-year UCL undergraduate CS course "COMP0005: Algorithms" [1] covers widely used algorithms that solve the problem of sorting. For some of these algorithms, e.g. "Merge Sort", it is difficult – if not impossible – to grasp their functionality intuitively from reading a description in pseudocode. In the lecture slides, the professor used animations to demonstrate them, greatly enhancing the students' understanding. Seeing elements being sorted in real time is a valuable tool in learning about these algorithms. Visualisation is of great importance for programmers, since it gives an insight into the practical steps that a sorting algorithm performs.

This need for a better understanding of sorting sparked the development of SACAT, Sorting Algorithm Complexity Analysis Tool. By validating, analysing and visualising students' implementations of sorting algorithms, SACAT will help them to understand the differences between theoretical and practical complexity analyses and also improve their programming skills.

1.2 Aim

The aim of this project is to provide a sorting algorithm analysis tool that can be used by anyone trying to analyse and visualise their sorting algorithm implementation in Python code. The main features include:

- Validating the user's implementation of a sorting algorithm
- Empirical testing on random, semi-random and predefined data sets of the algorithm
  - time complexity through timing and counting the number of operations,
  - space complexity by recording memory allocation
- Regression analysis and curve fitting to estimate the algorithm's big Oh and big Theta time and space complexity.
- Visualisation tools; in particular, plots showing the algorithm performance and bar charts showing the algorithm execution in real time.

2 Application Design

2.1 Program structure

The program will have a MVC (Model View Controller) architecture. The core logic is encapsulated in the Model, the GUI (Graphical User Interface) will be a part of the View and the Controller will regulate the communication between the two. This will allow easy addition of other views in the future, enabling a smooth migration of the entire program to a web or mobile app.

2.2 Resources

The program will be written in Python3, making use of the following additional libraries and modules:

- `random` *(built-in)* - Used for generating randomised tests to run on the input algorithm.
- `math` *(built-in)* - Used for performing advanced mathematical operations on test data and others.

- `time` *(built-in)* - Used for timing each test (to calculate time complexity) as well as to limit how long each test can run.
- `tracemalloc` *(built-in)* - Used for tracing the number of memory allocations and its usage.
- `ast` *(built-in)* - Used for building a custom profiler and adjusting parser.
- `matplotlib` - Used for plotting the results of testing the input algorithm and the results of analysing that data. Used for animating graphs with the matplotlib.animation submodule.
- `sklearn` - Used for analysing the data from the tests on the input algorithm. This module has built-in models for calculating Linear regression.
- `numpy` - Used for manipulating test data and reorganising it.
- `pyqt5` – Used for the GUI

2.3 GUI

The first version of the application will be a desktop app. One of the project deliverables is to develop a graphical user interface designed to make the interaction between the user and the Core simple and intuitive. The GUI is going to be implemented using PyQt5 (amongst other QT tools) and will be most likely a temporary shell of the app. Future plans include building a web application based on the Core (see section 5)

The interface will comprise of four main components:

- A text editor, in which the user will input their implementation either by typing it in or opening an existing one from an external file.
- A settings component, where the user will be able to specify the parameters of algorithm analysis such as t_{max}, T_{max} and choose which testing and analytic methods to use.
- A visualisation segment containing interactive complexity graphs for time and space as well as the bar charts reflecting the behaviour of the sorting implementation.
- An information tool responsible for showing all the essential information at each stage of the analysis, including error messages, successful tests and fit results.

The GUI will also provide options for saving the code and the results of analyses, such that multiple implementations can be benchmarked against different factors. This will be achieved by allowing the user to name their implementation, making its results distinguishable from others. Then each analysis can be visualised distinctly on the plot graphs (for instance by using a specific color and updating the legend).

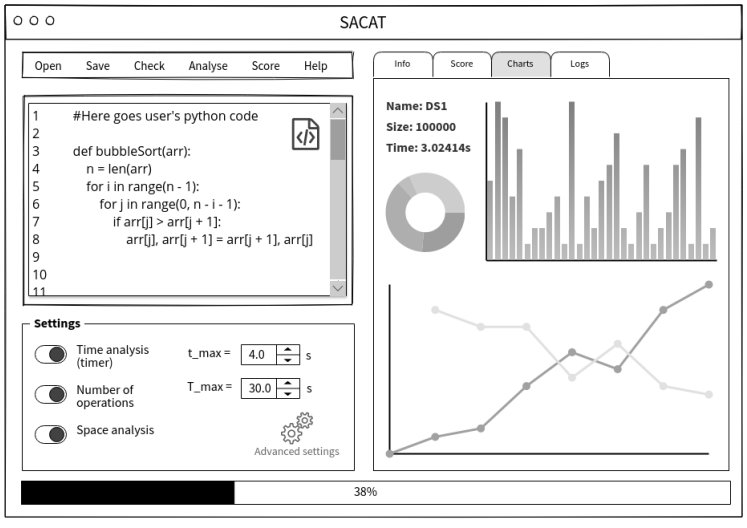


Figure 1: GUI Overview

3 Methodology of Analysis

The model will estimate an algorithm time and space complexity based on data that is gathered during run-time. In the data gathering process, the program runs *groups of tests* on the user's algorithm, each test producing a running cost (in terms of time, operations and memory) related to an input list type and length. After collecting all necessary data, they are analysed and used to estimate the complexity.

3.1 Time complexity based on time

One way of estimating the algorithm time complexity is to measure its *time of execution* for different input lists. A downside of this approach is that it may not provide very precise results. Even on the same machine, results produced by running the same test twice may vary because of interruptions caused by other processes running simultaneously. The resulting noise may negatively affect the user's ability to distinguish between the performance of different algorithms, which would defeat the purpose of this tool to some extent.

Nonetheless, estimating an algorithm's time complexity based on time plays a significant role in our design. The program is supposed to be a learning tool for a student. Therefore, including multiple ways of measuring performance can contribute to the student's understanding of algorithmic complexity. Crucially, this demonstrates the difference between the theoretical and real performance of algorithms.

3.2 Time complexity based on operation count

The most common definition of time complexity is the *number of operations* that an algorithm performs. The operation count is independent of other processes running on the machine, which makes it a more precise measurement method of time complexity than execution time. Furthermore, it is independent of the CPU and memory of the device running the algorithm.

To count operations, the user defined algorithm is firstly modified by an *adjusting parser*, which adds the code necessary for the analysis. A custom-built profiler groups operations by type (e.g. comparisons, assignments) and measures precisely how many times each operation type is performed. The profiler will recognise which lines in the user defined sorting algorithm contain operations affecting the time complexity and will be able to measure their impact during run-time.

3.3 Space complexity

Calculating the space complexity will be based on a similar process as the one described for the calculating the time complexity. To estimate the space complexity usage, peak memory usage will be recorded for each corresponding list length. Then, all gathered data will be passed to the data analysis module.

4 Core Functionality

The model consists of several stages. In the *Preliminary testing* stage, the user defined algorithm is tested for exceptions and correct functioning (e.g. sorting lists correctly). If the algorithm passes this stage, the *data gathering* process begins which runs the algorithm multiple times. While performing sorting, it is monitored by different run environments. After gathering an appropriate amount of data, the *data analysis* algorithms start and output their estimations. The general process is presented in Figure 2.

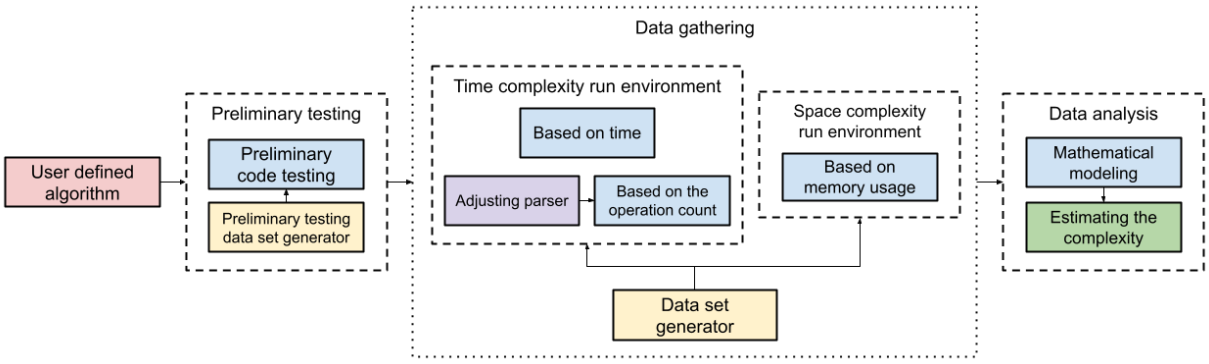


Figure 2: Core general structure

4.1 User input

An algorithm that the user inputs must obey several rules to be correctly analysed by the program:

- The algorithm must be executed through a function called `sort`, which takes a python list called `array` as its only parameter. This does not prevent the algorithm from having more than one function, since `sort` can call auxiliary functions.
- The algorithm must not rely on external libraries or imports from local files.
- The algorithm cannot modify or access any files or system settings. Besides, any such attempt will be blocked by the run environments.

4.2 Preliminary Testing

Before testing the complexity of the user defined sorting algorithm, the system must validate that the algorithm executes without errors and sorts lists properly. If the code throws exceptions, they must be handled by the program and the user should be warned that the code does not run properly. The first step here should be to run the user's code on a randomly generated list of 100 integers.

If there are no exceptions, the output list must be validated. This can be achieved by looping through the entire list and checking whether the first of two consecutive elements is equal or smaller to the second one. After this initial validation, the program will test the following corner cases:

- An empty list, `[]`
- A 1-element list, e.g. `[1]`
- A sorted list, e.g. `[1, 2, 3, ..., 100]`
- A sorted list in reversed order, e.g. `[100, 99, 98, ...,1]`
- A list containing equal elements, e.g. `[1, 1, 1, ..., 1]`
- A list of odd length, e.g. `[2,3,8,2,9]` and even length, e.g. `[2,9,1,3,9,4]`

When these tests are passed – i.e. no exceptions are raised – the algorithm is ready to be analysed for time and space complexity.

4.3 Data gathering

Data gathering plays a key role in estimating the complexity of an algorithm. The mathematical models described in section 4.4 require a sufficient number of data points to attain statistically significant results.

4.3.1 Time limits

Since the time that it takes an algorithm to sort a list can get arbitrarily long, it is important to run each test within an environment that keeps the time elapsed since the start of a test. If this time passes a certain threshold t_{max}, the test should be aborted and with it all tests that would have taken place in the future. The faster the algorithm, the more data could be collected within t_{max}, increasing the analysis' precision. This threshold value is modifiable by the user in the program settings.

Limiting the total number of tests to be performed would not be effective because sort algorithms can vary significantly in performance. Instead, the user can specify an upper bound on the time spent on all tests, T_{max}. This time limit will be distributed equally across all *groups of tests*.

4.3.2 Groups of tests

The length of the input list is not the only factor affecting the time complexity of a sort algorithm. Other factors that should be considered are, for example, the number of duplicates in the list or whether the list is already sorted. Therefore, *groups of tests* are specified and run separately. This enables the program to provide a more nuanced analysis, showing not only the estimated *average* time complexity, Big Theta, (i.e. performance on randomly generated lists), but also the *worst case* time complexity, Big Oh.

The following groups of tests are run:

1. Random lists – average case: lists consisting of elements randomly generated within a relatively wide range, such that it is unlikely that they contain many duplicates.
2. Duplicate lists – worst-case: lists consisting of elements randomly generated within a relatively tight range, such that they contain many duplicates. For some algorithms this may result in the worst time complexity.
3. Sorted lists – worst-case. For some algorithms this may result in the worst.
4. Reverse sorted lists – worst-case: sorted lists in reversed order. For some algorithms this may result in the worst time complexity.

The average case will be estimated using data from running a *Random lists* group of tests, and the worst case will be estimated by taking the worst calculated complexity using data from all groups of tests.

For each group of tests, lists of increasing length are generated by the program until time reserved for this group has elapsed. If needed, Python's `random` module is used to generate random numbers. The length of the lists in a group of tests will be increased exponentially using the formula:

$$S_i = 10^{0.1i},$$

where i is the number of test beginning with 0 and S_i is the number of elements in the list.

Although by default elements will be positive integers, user will be able to choose the type of elements in input lists from:

- positive integers,
- negative integers,
- integers,
- doubles.

4.4 Data analysis

As mentioned in the previous sections, the groups of data to be analysed are:

- time of execution,
- operations count,
- peak memory allocated.

These values (dependent variables) would be compared against lists of unsorted data with varying length (the independent variable). Each independent variable is plotted in several encodings against the input length:

- Polynomial: $y = x, y = x^2, y = x^3$,
- Linearithmic: $y = x \log x$,
- Exponential: $y = 2^x$,

where x is substituted with the independent variable (length of a list), and the resulting y plotted against it. The aim of these plots is to perform linear regression using the Ordinary Least Squares method – or different methods such as curve fitting or polynomial interpolation – on the data with a model supplied by the `sklearn` library. Then, the resulting correlation coefficients are used to estimate the tested algorithm's complexity class. The plots with the highest coefficients indicate the most likely complexities and they will be displayed to the user.

4.5 Example Analyses

4.5.1 Mergesort

This example shows how the method described in Section 4.4 would estimate Big Oh complexity of mergesort. The graph of time versus list length (Figure 3) shows the gathered test data. The closest fit based on the correlation coefficients below is $O(n \log n)$.

Methods used	Ordinary least Squares
Values measured	time
Testing lists lengths	from 1 to 1 000 000
Number of readings	50

$O(n)$ fit	0.9980142465341569
$O(n \log n)$ fit	**0.999183245476026**
$O(n^2)$ fit	0.9512071398181208

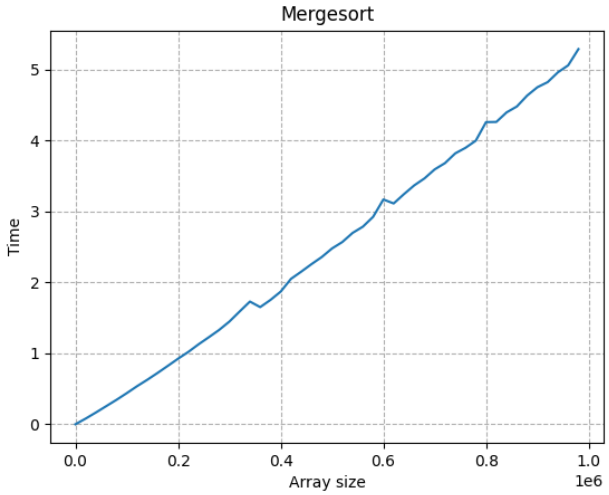


Figure 3: Mergesort data

4.5.2 Heapsort

This example shows how the method described in Section 4.4 would estimate Big Oh complexity of heapsort. The graph of time versus list length (Figure 4) shows the gathered test data. The closest fit based on the correlation coefficients below is $O(n \log n)$.

Methods used	Ordinary least Squares
Values measured	time
Testing lists lengths	from 1 to 1 000 000
Number of readings	50

$O(n)$ fit	0.996427980316863
$O(n \log n)$ fit	**0.998390222043668**
$O(n^2)$ fit:	0.9570196636456305

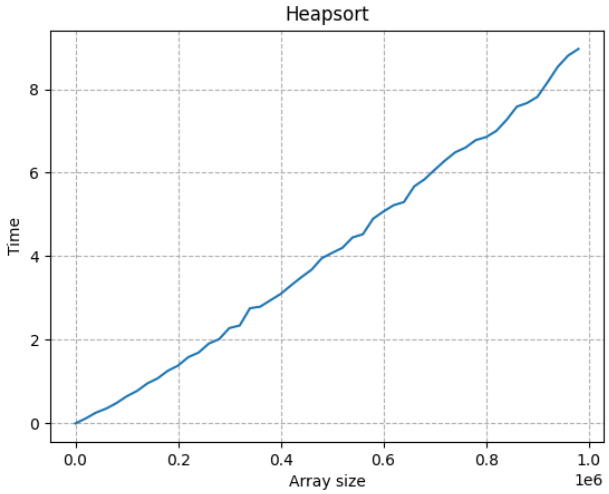


Figure 4: Heapsort data

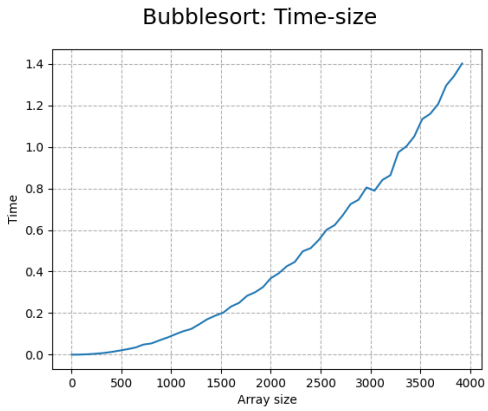
4.5.3 Bubblesort

This example shows how the method described in Section 4.4 would estimate Big Oh complexity of bubblesort. The graphs of time versus list length (Figure 5a) as well as operations versus list length (Figure 5b show the gathered test data. The closest fit based on the correlation coefficients below is $O(n^2)$ for both time and operations count analyses. Additionally, the graph in Figure 5b is noticeably smoother than the one in Figure 5a. This demonstrates that tests based on counting operations generate significantly less noise compared to tests based on timing.

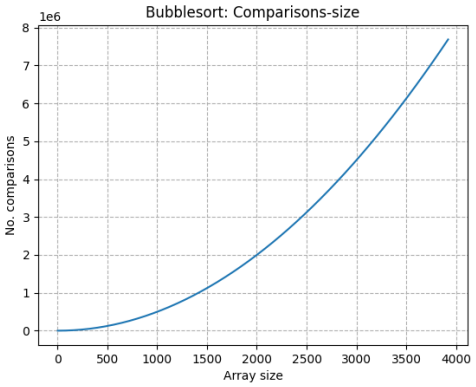
Methods used	Ordinary least Squares
Values measured	time & operations count
Testing lists lengths	from 1 to 4 000
Number of readings	50

$O(n)$ fit (time)	0.9330138172471819
$O(n \log n)$ fit (time)	0.9511589788294077
$O(n^2)$ fit (time)	**0.9988990483424111**

$O(n)$ fit (op.)	0.9352182627367351
$O(n \log n)$ fit (op.)	0.9532947615243036
$O(n^2)$ fit (op.)	**0.9999999960611927**



(a) Time based



(b) Operations count based

Figure 5: Bubblesort data

5 Possible Future Improvements

Due to time limitations, not all of the project possible features can be implemented, yet there is a number of possible future improvements to be developed. These include:

- Web deployment of the app to make it accessible to a wider audience.
- Analysis of algorithms based on user-defined data sets.
- A scoring feature to introduce competitiveness, which could motivate users to do more research on algorithms and experiment with the tool.
- Analysing algorithms operating on other data structures (not Python lists), e.g. trees, linked lists.
- Analysing algorithms that solve other problems, e.g. search.

References

[1] Module Lead Prof Licia Capra. "UCL Undergraduate Computer Science course COMP0005: Algorithms".

[2] Kevin Wayne Robert Sedgewick. *Algorithms, 4th Edition*. Addison-Wesley Professional, 2011.