

Divisualise! An Extensible, Interactive, Divide-and-Conquer Algorithm Visualiser

Armand Coretchi



4th Year Project Report
Computer Science and Mathematics
School of Informatics
University of Edinburgh

2024

Abstract

This report delineates the design and implementation of Divisualise, an extensible, interactive divide-and-conquer algorithm visualiser. This project aimed to improve upon existing divide-and-conquer (DAC) algorithm visualisations by creating a new tool, which provides users with a high-level view of the execution of various DAC algorithms alongside detailed information about the decisions made at each step.

Divisualise models DAC algorithms as trees of recursive calls with independently executable 'divide' and 'combine' phases. The platform provides a remarkably simple, yet powerful, API that allows implementors to create new visualisations effortlessly. Divisualise takes the naive recursive implementation of any DAC algorithm and handles the intricacies of input handling, presentation, and optimisations such as memoisation, to create a beautiful, interactive demonstration of its execution. Its implementation integrates modern web technologies, established design principles, and innovative visualisation techniques to create a responsive, accessible user interface.

A user survey conducted to evaluate Divisualise demonstrates that the platform has largely succeeded in its mission, with the majority of respondents finding the application easy to use, intuitive, and informative. The survey results also highlight areas for improvement, which will be addressed in future iterations of the platform.

We hope to continue the development of Divisualise and foster an open-source community around the project. By encouraging contributions from educators, students, and developers, Divisualise has the potential to grow into a comprehensive platform for learning about divide-and-conquer algorithms.

Research Ethics Approval

This project obtained approval from the Informatics Research Ethics committee.

Ethics application number: 778012

Date when approval was obtained: 2023-10-24

Participants who answered the Divisualise user survey were informed of this project's ethics application number and approval in the introduction to the online survey. No formal consent was collected, as the survey did not gather personally identifying information about the respondents.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(Armand Coretchi)

Acknowledgements

I would like to thank my wonderful supervisor, Professor Murray Cole, for his invaluable guidance throughout this project.

Table of Contents

1	Introduction	1
2	Background	3
2.1	Divide-and-Conquer Algorithms	3
2.2	Dynamic Programming and Memoisation	4
2.3	Algorithm Visualisation as a Pedagogical Tool	4
2.4	User Interface Design	6
2.5	Software Tools for Visualisations	7
2.5.1	SvelteKit	7
2.5.2	Tailwind CSS	8
2.6	Prior Work	8
2.6.1	Algorithm Visualizer	8
2.6.2	VisuAlgo	8
3	Designing Divisualise!	10
3.1	User Segmentation	10
3.2	Representing DAC Algorithms	11
3.2.1	Representing Recursive Calls	12
3.2.2	Inspecting Subproblems	13
3.2.3	Playback and Camera Controls	15
3.2.4	Dynamic Programming	16
3.3	Custom Input Data	18
3.4	The Visualised Algorithms	19
3.5	Aesthetic Considerations	20
3.6	Accessibility	21
3.6.1	Responsive Design	21
3.6.2	A Primer on Divide-and-Conquer Algorithms	22
3.7	Distribution	22
4	API Design	23
4.1	Type Safety	23
4.2	Modelling Input and Output Values	24
4.2.1	Sets of Named Values	26
4.3	Modelling Recursive Calls	26
4.3.1	Modelling the State of a Recursive Call	26
4.3.2	Recursive Calls and Recursive Cases	27

4.3.3	Memoisation for Free	29
4.3.4	Interacting with the Call Tree	30
4.3.5	Providing Details	32
4.4	Configuring New Algorithms	33
5	Frontend Implementation	34
5.1	Input and Output Values	34
5.2	The RecursiveCall Component	34
5.3	The Divisualise Component	35
6	User Evaluation	36
6.1	Survey Design	36
6.2	Quantitative Insights	37
6.3	Qualitative Feedback	37
7	Conclusions	38
	Bibliography	40

Chapter 1

Introduction

Divide-and-conquer (DAC) is an algorithmic design paradigm, employed by numerous widely-used algorithms, which solves problems by recursively breaking them down into smaller subproblems, solving these subproblems independently, then combining their solutions to solve the original problem. Despite being a fundamental concept in computer science (CS) curriculums, the abstract and recursive nature of DAC algorithms can make them challenging for students to comprehend, and for educators to teach effectively.

Visualisation has long been recognised as a powerful tool for demonstrating complex concepts, and algorithm visualisation (AV) tools—software designed to graphically represent the workings of algorithms—have been developed for educational purposes since the '80s [Brown and Sedgewick, 1984]. However, existing AV tools often struggle to demonstrate the elegance and simplicity of the DAC approach by emphasising low-level details, failing to convey the high-level recursive structure of DAC algorithms, and not providing sufficient interactivity for users to explore the algorithms' behaviour. In this report, we present Divisualise, an extensible, interactive divide-and-conquer algorithm visualiser designed to address these limitations by providing users with a high-level view of the execution of various DAC algorithms alongside detailed information about the decisions made at each step. Divisualise aims to be intuitive, engaging, and accessible to users with varying levels of expertise in computer science, from novice students to experienced educators.

We begin, in chapter 2, by introducing the notions of divide-and-conquer algorithms and dynamic programming. We then discuss the history of AV in CS education, noting the associated challenges and identifying best practices for the creation of AVs. Next, we note important considerations for designing user interfaces and introduce the reader to tools used to create software visualisations. Finally, we conclude the chapter by critically evaluating how two existing AV tools demonstrate the action of DAC algorithms.

Next, chapter 3 guides the reader through the design and feature set of Divisualise. We identify three target user segments to serve as the bedrock for our design decisions, then detail how we chose to visually represent the high-level structure of DAC algorithms while providing a mechanism for users to understand the intricacies of each step in

their executions. We discuss all of the interactive features we developed to create a comprehensive learning experience, and also describe the aesthetic and accessibility considerations made. Finally, we describe how we distributed Divisualise.

The two following chapters, 4 and 5, detail the implementation of the Divisualise API and frontend respectively. Chapter 4 focuses on the design of the API, which aims to provide a simple, extensible, and elegant interface for creating divide-and-conquer algorithm visualisations. We discuss the key components of the API, including the modelling of input and output values, recursive calls, and the configuration of new algorithms. Chapter 5 explores the frontend implementation, highlighting the design decisions and techniques employed to create an intuitive, responsive, and accessible user interface, such as the dynamic rendering of input and output values, the recursive rendering of the call tree, and the responsive layout that adapts to different screen sizes.

Chapter 6 presents the results of a user evaluation survey conducted to assess the effectiveness of Divisualise in meeting our design goals and to identify potential areas for improvement. We discuss the survey design, insights gained from the responses, and the criticisms and suggestions provided by the users.

Finally, in chapter 7, we summarise the key achievements of the Divisualise project, discuss the potential for future development, and highlight the importance of fostering an open-source community around the platform. We conclude by reflecting on the significance of Divisualise in the realm of algorithm visualisation and its potential to revolutionise the way divide-and-conquer algorithms are understood and appreciated by students and educators alike, ultimately contributing to the advancement of computer science education.

Chapter 2

Background

2.1 Divide-and-Conquer Algorithms

Divide-and-conquer (DAC) is an algorithmic design paradigm which solves problems by recursively breaking them down into smaller subproblems, solving these subproblems independently, then combining their solutions to solve the original problem; it is a foundational concept in algorithmic design which students often encounter early in CS education.

[Cormen et al., 2009] identifies three key stages of execution which are common to all DAC algorithms:

Divide the problem into smaller instances of the same problem

Conquer by recursively solving these sub-problems, or by directly solving a sub-problem if it is small enough.

Combine the results of these sub-problems to find a solution to the original problem.

Divide-and-conquer algorithms often provide elegant, efficient solutions to complex problems. Some examples of commonly used DAC algorithms are Merge Sort, Quick Sort, Binary Search, and Strassen's Algorithm. The applications of DAC algorithms span a plethora of domains including numerical linear algebra, computational geometry and data sorting; this ubiquity makes DAC algorithms a crucial part of every computer science curriculum - a deep, intuitive understanding of the paradigm is crucial for solving a large range of computational problems.

Due to their abstract, recursive, and often complex nature, students often have difficulties conceptualising the workings of DAC algorithms and the underlying principles of the DAC approach, underscoring a need for effective pedagogical tools which can make this challenging concept more digestible.

2.2 Dynamic Programming and Memoisation

Dynamic programming (DP) is an optimisation technique that can be applied to certain divide-and-conquer algorithms to improve their efficiency. The essence of dynamic programming lies in ensuring each subproblem is solved only once and storing its solution for future reference, thereby avoiding redundant computations [Cormen et al., 2009].

Dynamic programming algorithms are commonly implemented with a bottom-up approach, in which the solutions to subproblems are computed and stored systematically, building towards a solution for the root problem. An alternative approach to dynamic programming is memoisation, which can be applied to naive recursive implementations of divide-and-conquer algorithms. Memoisation involves augmenting naive recursive algorithms with a lookup table. When the algorithm encounters a subproblem, it first checks the lookup table to see if the subproblem has already been solved. If it has, the result is simply returned from the table. If the subproblem has not been previously solved, the algorithm computes the solution, stores it in the table, and returns the result.

Dynamic programming techniques are common topics in undergraduate computer science curriculums and software engineering interviews. An interactive means of visualising the effect of these techniques on DAC algorithms could prove highly valuable to educators, students, and jobseekers in software engineering.

2.3 Algorithm Visualisation as a Pedagogical Tool

Algorithm visualisation (AV) for Computer Science (CS) education has its origins dating back to the early '80s with tools such as the Balsa system [Brown and Sedgewick, 1984]. These visual systems have since evolved, particularly with the advent of modern web technologies, to find a significant place in Computer Science education. This section delves into the prevalence of visualisations in contemporary Computer Science curriculums, the effectiveness of these visualisations in aiding comprehension, and methods to measure this effectiveness, alongside exploring the attributes that constitute a compelling visualisation.

As early as 2003, educators had seemingly reached a consensus regarding the significant value of AVs in CS education, however, surveys have indicated that only half of all educators incorporate them in their teaching practices [Fouh et al., 2012]. [Fouh et al., 2012] identifies two primary challenges for teachers: finding suitable AVs and integrating them seamlessly into the curriculum. Since the publication of these findings, advancements in web technologies have considerably alleviated the difficulty of building and distributing algorithm visualisations, offering a more straightforward way to address the aforementioned issues. Additionally, the shift towards online or hybrid education in the post-COVID era has normalised students' engagement with online resources, potentially fostering a more conducive environment for adopting AVs in CS education.

Despite the prevailing consensus, a meta-analysis by [Hundhausen et al., 2002] reveals that the effectiveness of AVs for enhancing comprehension is not quite as clear-cut. The analysis covered 24 experimental studies, with only 11 showing a statistically

significant difference in outcomes between the groups using an AV tool and those using either a different tool, a different configuration of the same tool, or no AV tool at all. Notably, [Hundhausen et al., 2002] found that the differences in student outcomes were more significantly attributed to how students used the tools, rather than what the tools presented to students. In light of this, the task of measuring the effectiveness of AVs becomes crucial yet challenging. [Naps et al., 2003] employs Bloom's taxonomy, a framework delineating the different levels of learners' understanding ranging from fact-recall to evaluation, as a key measure for evaluating the effects of AV on students' understanding of algorithms. [Naps et al., 2003] also details the factors one might consider in evaluating understanding, being: progress, drop-out rate, learning time, and learner satisfaction. The challenges associated with measuring students' understanding stem from the dual nature of algorithms, they are profoundly conceptual yet fundamentally rooted in implementation. This duality makes it intricate to assign a particular 'level' of understanding, as students might grasp different facets of an algorithm to varying extents. The understanding of algorithms is multi-dimensional, encompassing not only theoretical comprehension but also practical application.

It is important to consider the particular qualities that make a good AV. [Naps et al., 2003] presents 11 best practices as commonly accepted suggestions drawn from experience which can be summarised as follows:

Interpretation Aids: Provide resources to help learners interpret the graphical representation.

Adaptability: Tailor the complexity of the visualisation to the knowledge level of the user.

Multiple Views: Offer various views of an algorithm, like control flow or data structure state.

Complexity Information: Include efficiency analysis data to elucidate the performance characteristics of the algorithm.

Execution History: Offer a history feature to help learners track previous steps.

Flexible Execution Control: Allow flexible control of the AV execution.

Learner-built Visualisations: Encourage learners to build their own visualisations to deepen understanding.

Custom Input Data: Allow learners to specify their own input data.

Dynamic Questions: Incorporate interactive questions at intervals to encourage reflection.

Dynamic Feedback: Provide real-time feedback on learners' interactions within the visualisation.

Complementary Explanations: Accompany visualisations with clear explanations.

2.4 User Interface Design

The design of user interfaces and the overall user experience (UX) play a crucial role in the effectiveness of any application. Research has shown that the aesthetic appeal of an interface can significantly influence users' perception of its usability and their overall satisfaction with the tool. [Tractinsky et al., 2000] found that users perceive aesthetically pleasing interfaces as more usable, even when the actual usability remains unchanged. This phenomenon, known as the aesthetic-usability effect, highlights the importance of considering visual appeal in interface design to enhance user engagement and potentially improve learning outcomes.

In addition to aesthetics, several key principles guide the design of effective user interfaces. [Blair-Early and Zender, 2008] identifies ten interface design principles and four general design principles that contribute to creating intuitive, user-friendly, and engaging interfaces. The interface design principles include:

Obvious Start: Providing a clear starting point for user interaction.

Clear Reverse: Offering an evident way to undo actions or exit the interface.

Consistent Logic: Maintaining internal consistency in content, actions, and effects.

Observe Conventions: Respecting familiar interface conventions and user expectations.

Feedback: Providing tangible responses to user actions.

Landmarks: Offering reference points for context and navigation.

Proximity: Grouping related elements and minimizing the distance between content and interface.

Adaptation: Allowing user customization and adapting to user needs.

Help: Providing readily accessible assistance when needed.

Interface Is Content: Integrating interface elements with content to minimize distraction.

The four general design principles encompass:

Subject Matter: Making the subject matter obvious from the start.

Interface Visualization: Using visual form apt to the content to embody the interface.

Content + Form: Designing visually engaging interfaces that reflect the nature of the content.

Metaphor: Employing metaphors to introduce new or obscure concepts, particularly in narrative-based content.

While the impact of UI/UX design on learning outcomes in the context of algorithm visualisation tools requires further research, it stands to reason that more usable interfaces may lead to improved understanding and retention of the presented concepts. By reducing cognitive load and minimizing distractions, well-designed interfaces allow

users to focus on the educational content. Moreover, the aesthetic-usability effect suggests that visually appealing interfaces may encourage users to engage with the tool for longer periods, potentially leading to deeper exploration and comprehension of the algorithms being visualised.

2.5 Software Tools for Visualisations

Software visualisations (and GUI applications) now fall into one of two categories: native applications and web applications. Native applications are executables which run directly on a user's computer, whereas web applications are built with technologies such as JavaScript, HTML and CSS and execute within a web browser. Historically, native applications have been preferred for complex visualisations due to their robust performance and hardware access, however recent advancements in web technologies have significantly bridged this gap. Features of the present web, such as the HTML Canvas API and CSS animations, now enable intricate graphical rendering and interactive visualisations within web browsers. The inherent accessibility of the web - requiring no installations and providing a simple means of global access - presents a compelling advantage over native applications, making it an attractive choice for developing sophisticated and accessible algorithmic visualisations.

2.5.1 SvelteKit

SvelteKit [Svelte contributors, 2024] is a web framework that leverages Svelte, a modern programming language designed for creating web-based GUI applications. Svelte provides an elegant API and acts functionally as a super-set of JavaScript, offering enhanced functionality and intuitive APIs for interactivity while retaining JavaScript syntax familiar to all web developers. Through its reactive programming model, developers can construct intuitive, reactive user interfaces. Svelte code is compiled into highly optimized, imperative JavaScript that directly manipulates the DOM, eliminating the need for older, more resource-hungry techniques such as a virtual DOM.

SvelteKit emerges as an increasingly popular option in a crowded ecosystem of web design tooling. Some alternatives and competitors are:

React: The most widely used web reactivity framework, known for its pioneering use of the 'virtual DOM'.

Angular: A comprehensive framework backed by Google, Angular is a precursor to React with dwindling adoption.

Vue.js: A lesser-used, but powerful alternative to React providing a similar API.

D3.js: Especially suited for dynamic, interactive data visualisations in web browsers, leveraging modern CSS features and HTML Canvas.

2.5.2 Tailwind CSS

Tailwind CSS [Tailwind CSS contributors, 2024] is a utility-first CSS framework that enables the rapid development of intricate, responsive web user interfaces. Unlike other CSS frameworks which provide specialised classes for styling certain elements, Tailwind CSS exposes a plethora of low-level utility classes which enable fine-grained control of an interface's appearance directly in the markup. This approach promotes design consistency, reduces the need for custom CSS, and results in smaller CSS files and faster load times. Tailwind CSS also offers utility classes for responsive design, allowing developers to specify different styles for various screen-size breakpoints and create an interface that remains consistent and accessible across a wide range of devices.

2.6 Prior Work

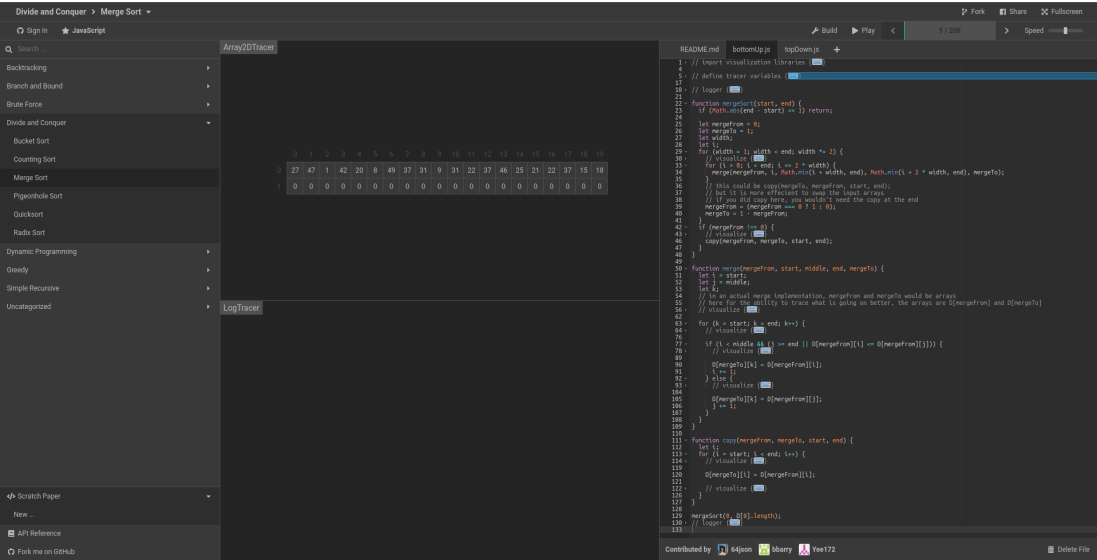
Over time, a multitude of algorithm visualisation tools for educational purposes have been developed; Older AVs such as BALSA [Brown and Sedgewick, 1984] were built as native apps whereas the majority of modern tools are web-based. Two popular, contemporary, web-based examples are detailed below. Existing visualisations, particularly focusing on DAC algorithms, are low-level and sequential. The algorithms are stepped through and show the operations on the underlying data structures at each step, lacking the ability to explore the different 'branches' of a recursive problem independently and perhaps failing to wholly communicate the nuance and elegance of the design choices made for the algorithms in question.

2.6.1 Algorithm Visualizer

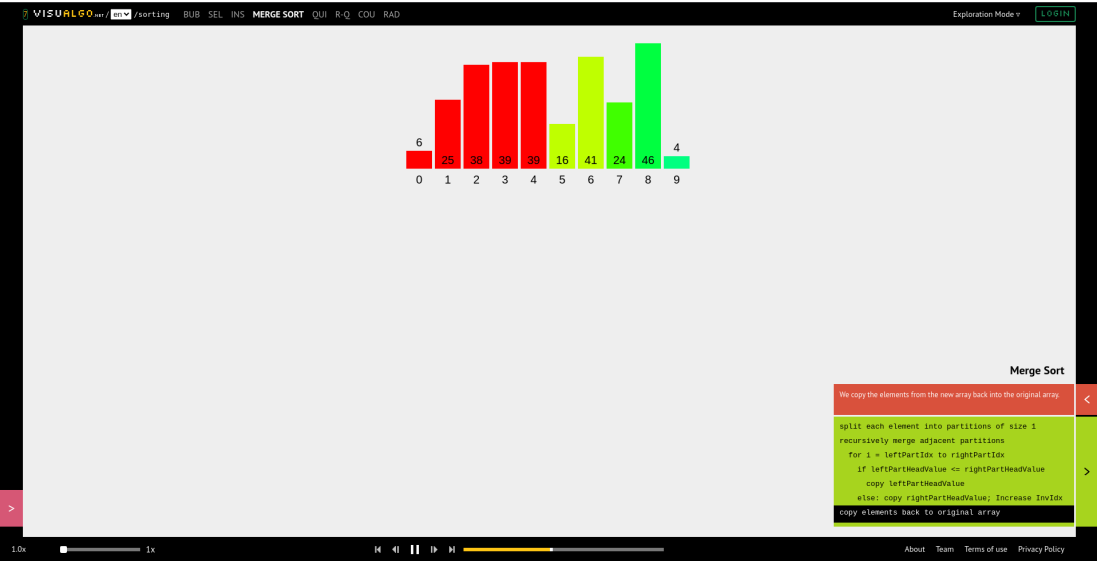
Algorithm Visualizer is a web-app built with React that provides a means of stepping through the code and execution of various algorithms whilst simultaneously providing a visual representation of the underlying data structure that the algorithm is operating on with a choice of views. As this tool is based on stepping through real code, there is no fine-grained control over the branches of execution explored when visualising a DAC algorithm, as demonstrated in 2.1a.

2.6.2 VisuAlgo

VisuAlgo is a web app and interactive educational platform created in 2011 to foster a deeper understanding of data structures and algorithms. It provides unique, interactive visualisations for a multitude of complex algorithms. The platform currently features 24 visualisation modules, with ongoing efforts to develop more visualisations; each visualisation module in VisuAlgo also now includes its own online quiz component, by which learners can self-assess their understanding. Unlike Algorithm Visualizer, which centres on stepping through real code, VisuAlgo offers a varied range of visualisations alongside a built-in quiz system, promoting a self-paced and thorough learning and assessment experience.



(a) Algorithm Visualizer



(b) VisuAlgo

Figure 2.1: Two existing visualisation tools demonstrating the operation of MergeSort

Chapter 3

Designing Divisualise!

In this chapter, we present the design and feature set of Divisualise, an extensible, interactive divide-and-conquer algorithm visualiser. Divisualise is implemented as a web application, utilising SvelteKit, TypeScript, and Tailwind CSS; more details on its implementation can be found in chapters 4 and 5.

We begin by identifying and analysing three target user segments, each with distinct needs and levels of expertise, which serve as the foundation for our design decisions. Following this, we detail our approach to visually representing DAC algorithms, the interactive features we built to facilitate a comprehensive learning experience, and the aesthetic and accessibility considerations made in the design process.

3.1 User Segmentation

In designing Divisualise, we identified three primary target user segments, each with distinct needs.

The first group consists of teachers using Divisualise as a teaching aid in their lessons. They may instruct students to use the tool independently or employ it to present the operation of DAC algorithms to a class. Teachers may wish to create visualisations for DAC algorithms not yet implemented in Divisualise.

The second group comprises advanced students, already familiar with the general concepts of DAC algorithms. These students will seek to deepen their understanding of specific algorithms and their implementations. They require a tool that provides detailed insights into the workings of the algorithms and allows for interactive exploration. Advanced students may also wish to implement their own visualisations to aid their learning, and enabling this action is considered a best practice by [Naps et al., 2003].

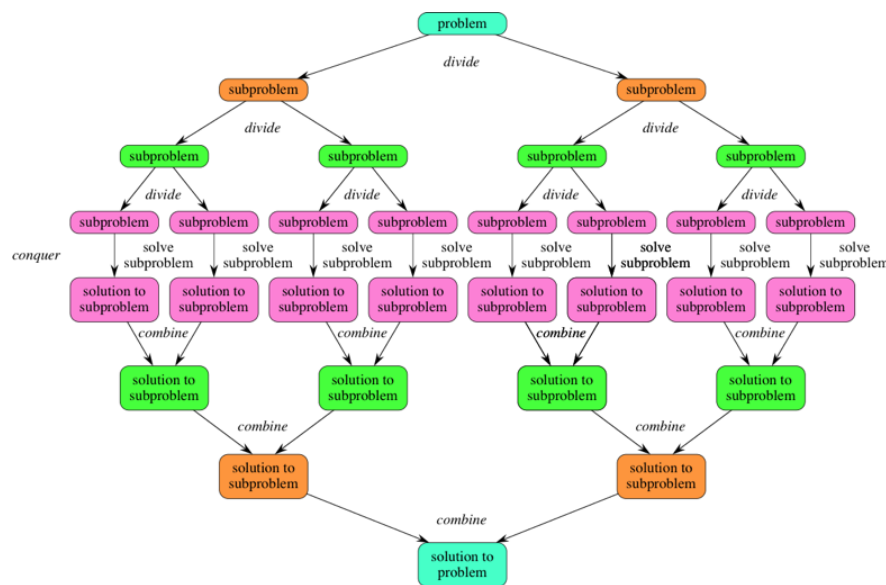
The third group includes students unfamiliar with, or with limited understanding of, the DAC paradigm. These students need a tool that presents the concepts in an accessible and intuitive manner, guiding them through the fundamental principles and problem-solving strategies employed in DAC algorithms from the ground up.

Considering the diverse needs of these user segments, we aimed to create an interface

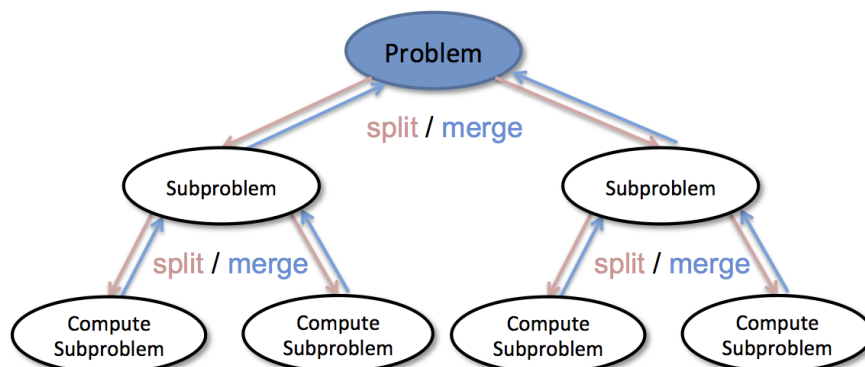
that caters to both novice and advanced learners, providing flexibility for those who wish to tweak, or create, visualisations.

3.2 Representing DAC Algorithms

We sought to create an intuitive, interactive visual representation of DAC algorithms that aligns with their natural, recursive structure. Existing visual aids (Figure 3.1) most commonly present DAC algorithms as trees of problem input and outputs, with the tree growing (downwards by convention) as the root problem is recursively divided into subproblems. We thought it apt to present DAC algorithms similarly in Divisualise, allowing teachers to augment their existing materials without introducing additional cognitive overhead for students with a new visual representation.



(a) A representation presenting subproblems being 'combined' downwards.



(b) A representation showing subproblems as being 'merged' upwards.

Figure 3.1: Two images from the first page of a Google image search for 'divide and conquer algorithms'.

Figure 3.1 shows two different representations of DAC algorithms. The first represen-

tation depicts subproblem solutions as being 'combined' downwards, below the base cases. The second representation illustrates subproblem solutions being combined back into their 'parent' problems, moving upwards.

We believed the second representation more accurately reflected the structure of a recursive call tree, with each node corresponding to a single recursive call, and that the prevalence of the first representation likely stems from the limitations of static media in effectively conveying the upward combination process.

Considering these factors, we designed an application that allows users to explore the execution of a DAC algorithm by interacting with its recursive call tree. In our visualisation, subproblem results bubble upwards towards the root as subproblems are solved, mimicking the actual flow of a recursive algorithm.

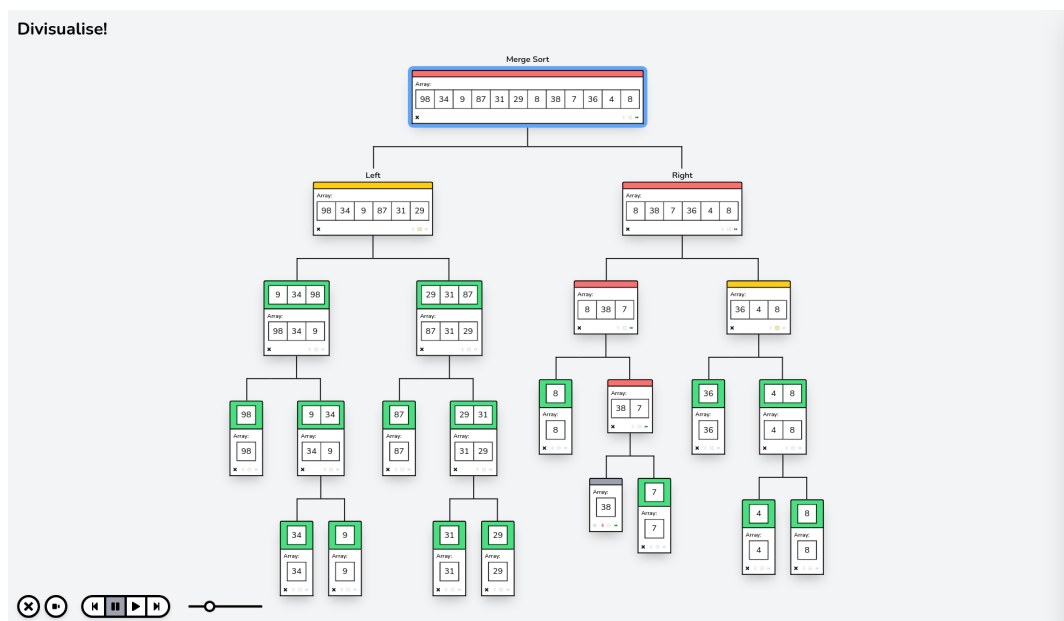


Figure 3.2: Divisualise displaying a partially solved Merge Sort call tree

3.2.1 Representing Recursive Calls

Each node in the Divisualise tree corresponds to precisely one recursive call, which can be in one of the following states, mirroring the stages of execution identified by [Cormen et al., 2009]:

Undivided The problem has not yet been split into subproblems; the call is necessarily a leaf node in our tree.

Divided The problem has been divided into subproblems, which are displayed as children of this call in the tree. The call may still be a leaf node if it corresponds to a base case.

Solved All subproblems (if any) are solved, and their results have been combined to find the solution to the current call.

As shown in Figure 3.2, nodes of all states display their input values, similar to the representations in Figure 3.1. Additionally, solved nodes present their result above the input, demonstrating how subproblem solutions bubble up the call tree. We use colours to indicate the state of each node: grey for undivided, red for divided, and green for solved. Furthermore, we visually differentiate 'combinable' calls (i.e., divided calls with all subcalls solved) by colouring them yellow. Our use of colour to convey call state adheres to the 'Feedback' interface design best practice identified by [Blair-Early and Zender, 2008]; moreover, our deliberate choice of traffic light colours follows the 'Metaphor' best practice, aiming to leverage users' existing associations with these colours: red for unsolved (stop), green for solved (go), and yellow for 'combinable' (prepare to go), representing an intermediate state.

To facilitate navigation and inspection of the call tree, we provided users with a 'camera' that can pan, zoom in, and zoom out of any section of the tree, allowing for a clear view of both the overall structure and individual nodes. Figure 3.3 shows a close-up view of a recursive call in various states.

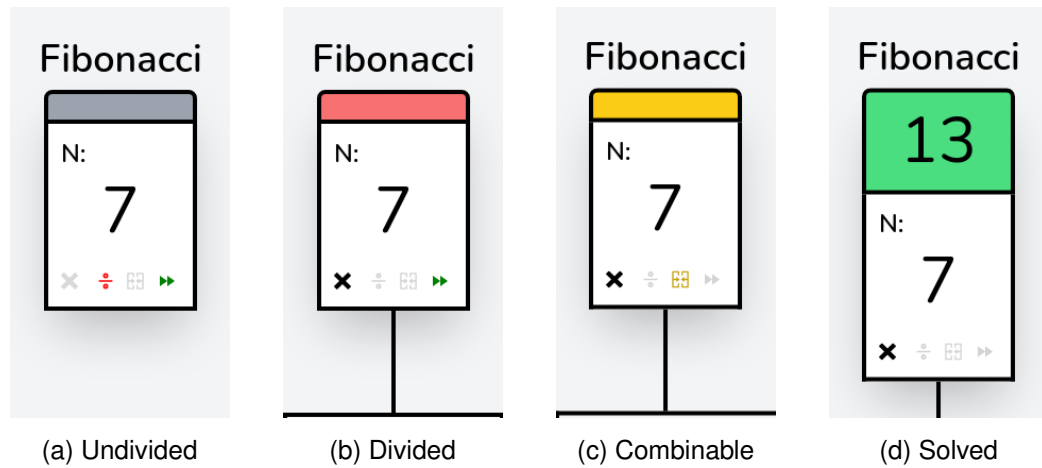


Figure 3.3: Close-up views of the same recursive call in different states.

Users can interact with individual calls using buttons to divide, combine, conquer, and reset them to an undivided state, when applicable. The buttons are coloured according to the state they lead to and are greyed out when unavailable, adhering to the 'feedback' best practice [Blair-Early and Zender, 2008]. This interactivity allows users to traverse the tree in order and revert any branch to an unsolved state, meeting the 'Flexible Execution Control' and 'Execution History' AV best practices identified by [Cormen et al., 2009]. By providing this level of control, Divisualise facilitates a deeper understanding of the recursive nature and problem-solving process of DAC algorithms.

3.2.2 Inspecting Subproblems

To improve upon existing algorithm visualisations, we sought to provide users with the ability to inspect the details of each subproblem as they navigate the recursive call tree. While the Divisualise call tree offers a valuable representation of the high-level structure and execution of recursive DAC algorithms, we wanted to give users access

to additional information about the decisions made at each stage of an algorithm's execution.

To achieve this, we introduced a 'details panel' that displays a slideshow explaining the action of a user-selected call based on its current state. Users can 'highlight' a call by clicking on it, which updates the details panel accordingly.

The slideshow consists of slides containing written explanations and intricate animations of the input values, illustrating the behaviour of the selected call. For example, highlighting a 'divided' Merge Sort call shows the input array being split into two subarrays, while selecting a 'solved' call demonstrates the merging of sorted subarrays.

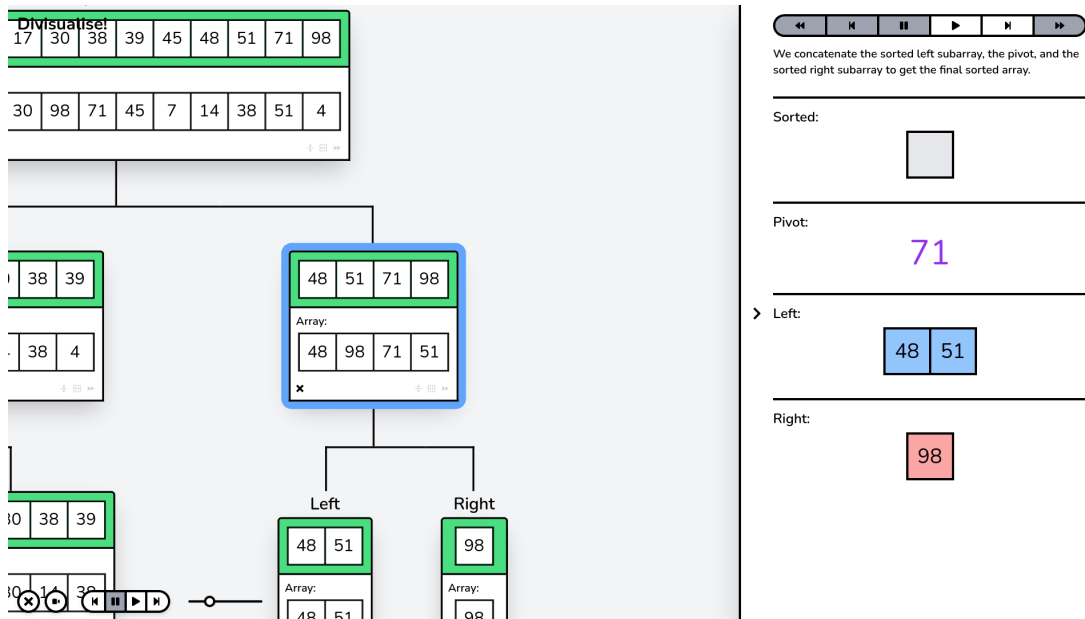


Figure 3.4: Subproblem details panel showing the merging of sorted subarrays in Quick Sort

The details panel (Figure 3.4) includes controls for users to play the slideshow as an animation, step forward or backwards through the slides, and control the animation progress within each slide. To accommodate different user preferences and learning needs, the details panel is collapsible, allowing users to focus solely on the call tree view if desired. This feature aligns with the 'Adaptability' best practice identified by [Blair-Early and Zender, 2008] by enabling users to tailor the interface to their needs.

Our slideshow model offers significant flexibility and supports several [Naps et al., 2003] best practices:

Complementary Explanations: The slideshows provide detailed explanations to accompany the visualisation.

Multiple Views: By presenting state information, the slideshows offer additional perspectives on the algorithm.

Complexity Information: Details about the algorithm's complexity can be included in the written explanations.

Interpretation Aids: The slideshows guide users through the algorithm’s execution, benefiting novice learners.

This approach improves upon the prior work we have discussed, as it enables the presentation of both a high-level structural overview, and the fine-grained details, of an algorithm. Furthermore, these slideshows are well-suited for teachers using Divisualise to present algorithms to a class. By providing users with the ability to inspect subproblem details, alongside the high-level call tree, Divisualise offers a more comprehensive and informative learning experience compared to existing algorithm visualisations.

We considered integrating ‘Dynamic Questions’ and ‘Dynamic Feedback’ into our slideshows but ultimately deemed these features to be outside the scope of this project which primarily focuses on the presentation aspect of algorithm visualisation. We foresee possible future extensions that leverage the details panel to question the user and provide interactive feedback, further enhancing the learning experience.

3.2.3 Playback and Camera Controls

While our call tree and details panel provide users with flexible control over the execution of the recursive call tree, allowing them to expand branches in arbitrary order, this flexibility may not always be desired. As recursive algorithms imply a specific order of execution, we enable users to traverse the call tree in order by providing playback controls.



Figure 3.5: Playback controls for stepping through the recursive call tree. The ‘camera lock’ button is hovered.

Figure 3.5 shows a close-up of our playback controls, which sit in the bottom-left corner of the camera view. The controls include buttons to step forward and backwards through an algorithm’s execution, alongside play/pause buttons to initiate and halt playback. During playback, the currently active call is highlighted, and if the details panel is expanded, the corresponding slideshow is synchronized with the execution progress. Users can also adjust the playback speed to their preference. Additionally, a reset button allows users to reset the entire call tree to its initial state, providing a convenient way to start the visualization anew.

To further enhance the user’s ability to follow the algorithm’s execution during playback, we introduced a camera lock feature that automatically pans the camera to centre on the currently highlighted call; this allows users to follow the execution without manually adjusting the camera. We foresee this feature being of particular value to teachers using Divisualise in a presentation setting, as it ensures the audience’s focus remains on the most relevant aspects of the algorithm’s execution.

The playback controls are designed to be tactile and responsive, highlighting and expanding on hover to provide visual feedback, which aligns with the 'feedback' user interface design best practice identified by [Blair-Early and Zender, 2008].

3.2.4 Dynamic Programming

Dynamic programming (DP) is an optimisation technique crucial for the efficient implementation of many DAC algorithms. However, students often struggle to gain an intuitive understanding of how DP techniques work, and how they reduce the time complexity of DAC algorithms. In light of this, we desired a feature that would provide an interactive, visual way to observe the impact of DP techniques. To drive this point home, we decided to implement the DP feature as a simple toggle button on visualisations for algorithms that support DP; this way, users can seamlessly switch between the naive and optimised versions of the algorithm, making the impact of DP immediately apparent.

We chose to focus on memoisation, a top-down approach to dynamic programming, where the results of previously computed subproblems are stored and reused when needed. Memoisation can be automatically applied to any naive recursive implementation without further consideration, making it an ideal candidate for our visualisation tool. To visualise memoisation in action, we extended our recursive call model with a new 'memoised' state, represented by blue nodes in the call tree. When a user enables memoisation, the call tree is dynamically updated to reflect the optimised execution of the algorithm. Subproblems that would be redundantly computed in the naive recursive implementation are instead resolved through memoisation, resulting in a visibly smaller call tree (Figure 3.7). This visual transformation provides a tangible representation of the efficiency gains achieved through dynamic programming.

The memoisation feature in Divisualise is designed to maintain a consistent state across the call tree. When a user interacts with the tree or enables memoisation, the tree automatically resolves dependent calls to ensure the correctness of the memoised results. Users can also transition back to the naive recursive version of the algorithm, with memoised calls expanding to reveal their underlying computations. This behaviour, demonstrated in Figure 3.8, adheres to the 'Consistent Logic' user interface design best practice identified by [Blair-Early and Zender, 2008]. Furthermore, when a user inspects a memoised or would-be memoised call in the details panel, Divisualise highlights the subproblem on which the call depends (Figure 3.9), providing users with a clear understanding of the dependencies between subproblems and how memoisation leverages these dependencies to avoid redundant computations.

The dynamic programming functionality in Divisualise aligns with several algorithm visualisation best practices identified by [Naps et al., 2003]. It provides 'Multiple Views' by presenting both the naive and memoised execution of an algorithm; it offers 'Complementary Explanations' through the details panel, which highlights the dependencies between subproblems; moreover, by allowing users to toggle memoisation on and off and observe the impact on the call tree, Divisualise provides 'Flexible Execution Control'. By visualising the reduction in the number of subproblems computed and the dependencies between subproblems, students can develop a deeper understanding of

how dynamic programming techniques optimise recursive algorithms.

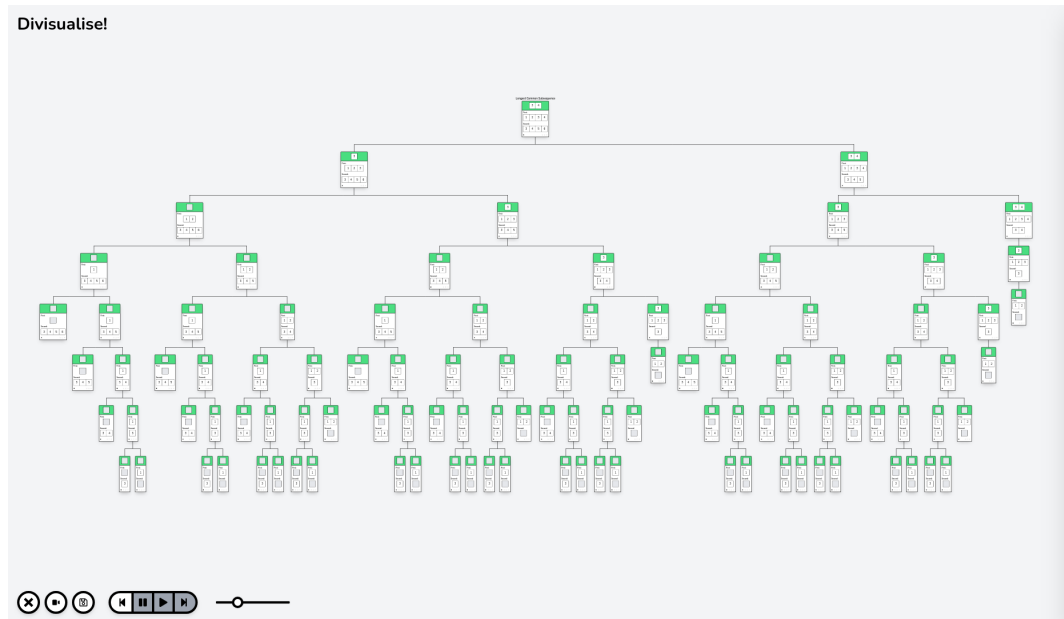


Figure 3.6: A solved call tree for the naive recursive Longest Common Subsequence algorithm

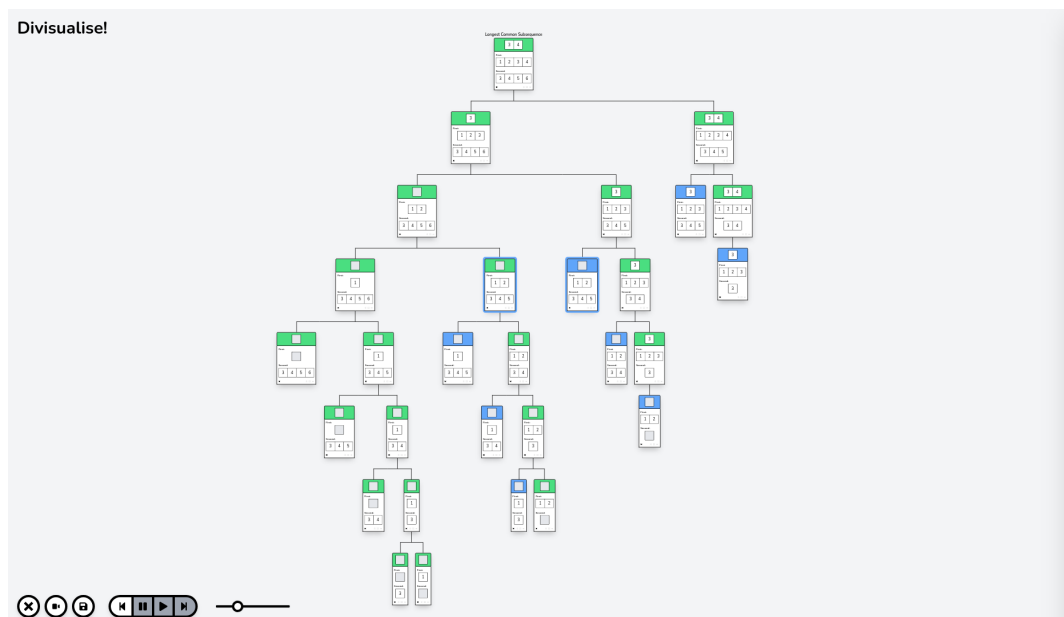


Figure 3.7: Toggling memoisation, the call tree shrinks as later overlapping subproblems transform into blue 'memoised' calls

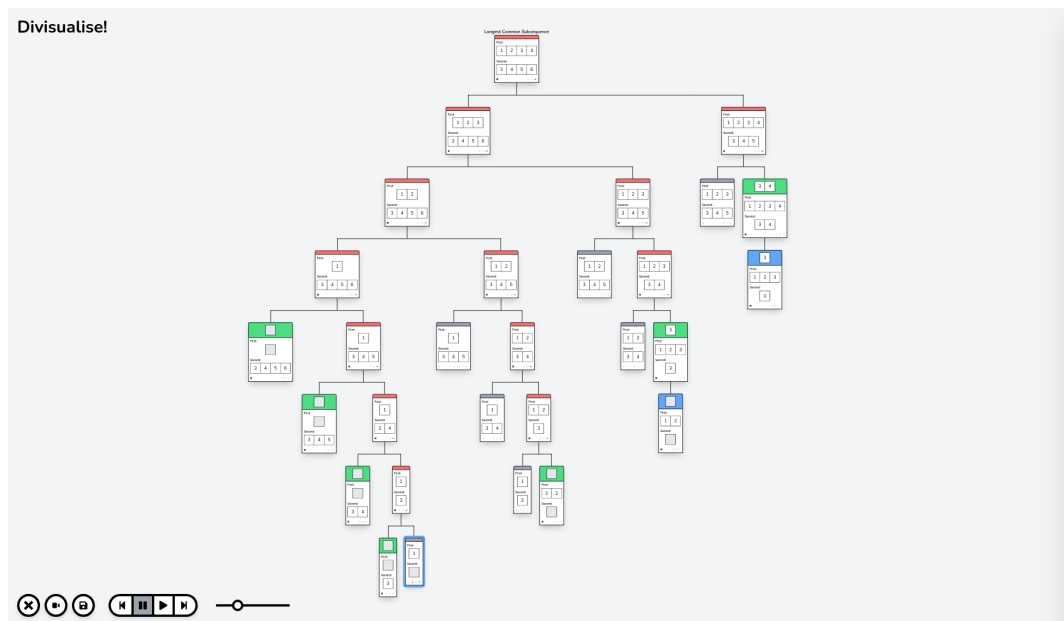


Figure 3.8: As we reset a base case our tree resolves dependent calls to maintain consistent state, unsolving out-of-order memoised calls

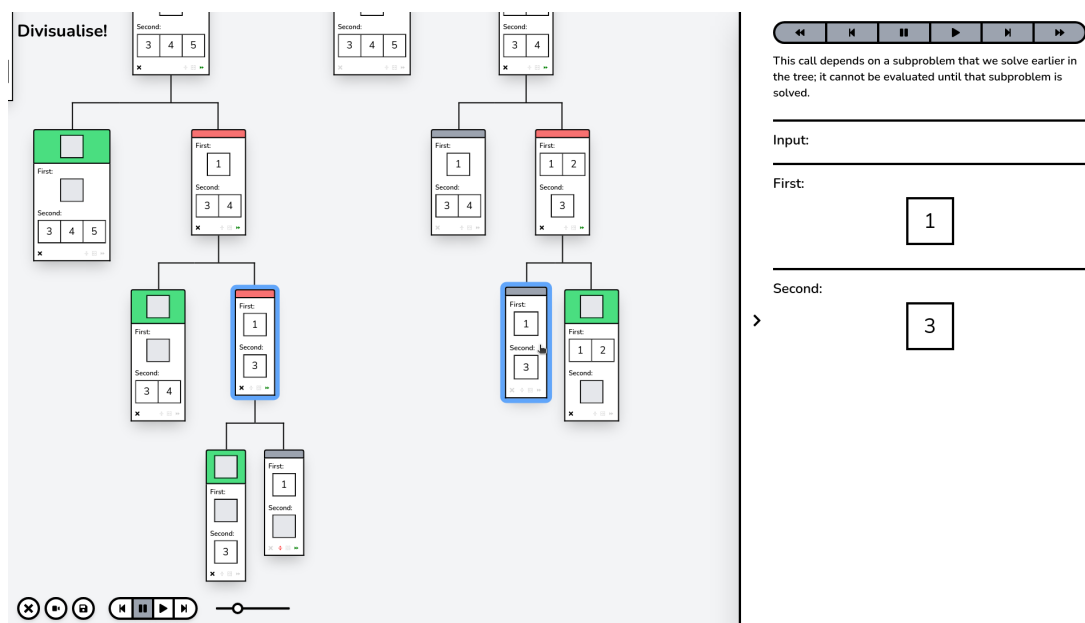


Figure 3.9: Inspecting a memoised, or would-be memoised, call highlights the subproblem on which it depends

3.3 Custom Input Data

The performance characteristics of certain DAC algorithms can vary significantly with the input provided; we thought it apt to follow the [Naps et al., 2003] best practice of 'Custom Input Data', allowing users to observe best, worst and average case behaviours in an intuitive, visual manner via the Divisualise call tree paradigm.

We designed custom input components (Figure 3.10) that aesthetically match the values presented in our visualisations, improving the overall cohesion and user experience of Divisualise. These components were crafted with ease of use in mind, eliminating the need for users to manually type comma-separated numbers or other complex input formats. Additionally, we limit the size of numbers in lists and matrices to ensure they are presented appropriately within the visualisation.

Binary Search

Array
Please enter a list of numbers

4	5	6	7	8	9	10
---	---	---	---	---	---	----

Target
Please enter a number

8

Submit

Cofactor Expansion

Matrix
Please input a matrix

Size: 4

4	5	10	11
0	3	6	9
0	0	2	8
0	0	0	1

Submit

Closest Pair of Points

Points
Please click on the square to input points.

Submit

(a) Array and Number inputs
(b) Matrix input
(c) Planar points input

Figure 3.10: Custom Divisualise input interfaces

3.4 The Visualised Algorithms

We sought to provide visualisations for a wide range of DAC algorithms, with the goal of enabling learners to understand the DAC approach independent of the implementation details of any particular algorithm. For teachers and advanced students who may wish to add, or modify, algorithms, we aimed to make the process as simple as possible; the "API Design" chapter delves into the details of how we achieved this goal, and the steps required to implement new algorithms.

To showcase the versatility of our visualisation framework, we implemented visualisations for the 10 following DAC algorithms:

- Merge Sort
- Quick Sort
- Closest Pair of Points
- Maximal Subarray
- Longest Common Subsequence (DP enabled)
- Fibonacci (DP enabled)
- Karatsuba's Algorithm
- Binary Search

- Strassen's Algorithm
- Cofactor Expansion

We chose these algorithms as they span a range of problem domains and work upon varied data types, allowing users to observe the DAC paradigm at work in many settings.

3.5 Aesthetic Considerations

In the design of Divisualise, significant effort was placed into creating an aesthetically pleasing user interface, with hopes that the aesthetic-usability effect [Tractinsky et al., 2000] may serve to improve the learning outcomes of our users. We leveraged colour palettes provided by Tailwind CSS, offering a harmonious set of colours that work well together. The interface employs a minimalist, high-contrast visual style characterised by stylish, thick black borders and hard edges. This design choice not only contributes to the overall aesthetic appeal but also enhances the clarity and readability of the interface elements.

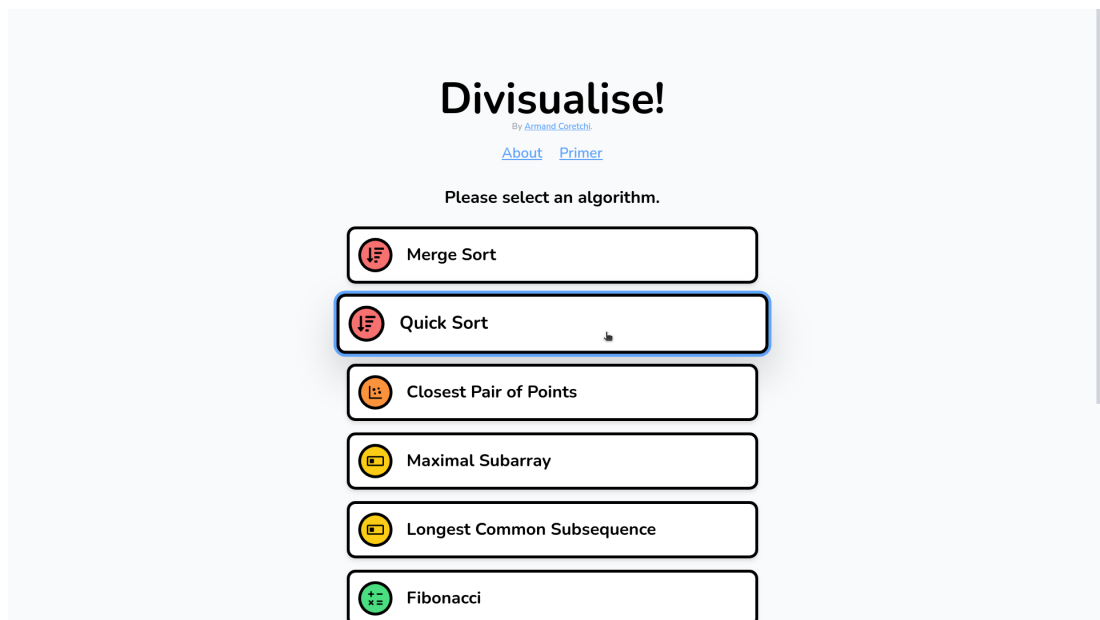


Figure 3.11: The Divisualise main menu.

The colour scheme of the interface is predominantly neutral, using colour strategically to draw attention to interactive elements and assist in the presentation of the execution of DAC algorithms. The neutral background allows the coloured elements, such as the call tree nodes and highlighted control buttons, to stand out and guides the user's attention to the most relevant parts of the interface.

The interface layout is clean and uncluttered, with ample whitespace to ensure the focus remains on the algorithm visualisation. We used an elegant, playful, sans-serif font called Nunito, licensed under the Open Font License, which contributes to the overall aesthetic and ensures readability across different screen sizes.

3.6 Accessibility

Accessibility was a key consideration throughout the design process of Divisualise - creating an inclusive educational tool benefits all users. We aimed to ensure that Divisualise could be effectively used by a wide range of students, including those with disabilities.

One of our primary goals was to create an interface that is fully navigable, and usable with, a keyboard, enabling users who cannot use a mouse to access all functionality. We ensured every interactive element could be reached and activated using the keyboard alone. Furthermore, we sought to make Divisualise compatible with screen readers, allowing visually impaired users to understand and interact with the visualisations; this involved structuring the content logically and using clear and descriptive labels for all interactive elements.

3.6.1 Responsive Design

As of 2016, the majority of web traffic in the USA comes from mobile devices [Google, 2024], underscoring the importance of creating responsive web applications that adapt to different screen sizes. We made use of Tailwind CSS responsive design features to ensure that Divisualise is accessible on a wide range of devices; Figure 3.12 demonstrates how the interface adapts to smaller screens.

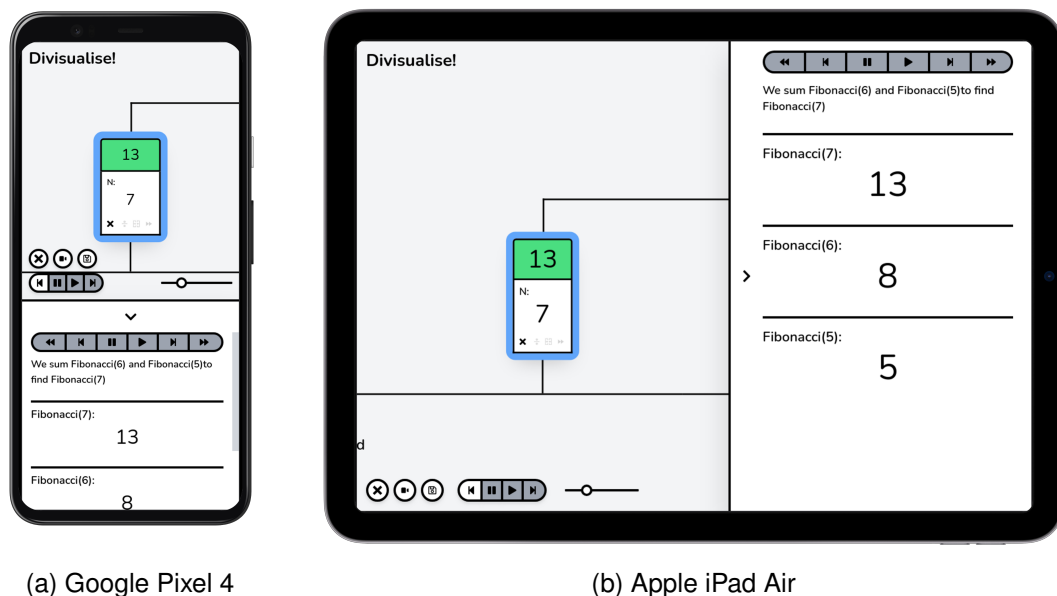


Figure 3.12: The Divisualise interface adapts to mobile devices.

[Department for Education, 2023] tells us that 54% of secondary schools in the UK provide tablet devices for student use in lessons. We anticipate that responsive Divisualise interface could be particularly valuable for teachers using mobile devices in the classroom. Students could use their own devices, or school-provided tablets, to follow along with a presentation or explore the DAC paradigm independently during lessons.

3.6.2 A Primer on Divide-and-Conquer Algorithms

To support students unfamiliar with, or with limited understanding of, the DAC paradigm, we included a primer on DAC algorithms, dynamic programming, and algorithmic complexity.

Accessible directly from the Divisualise main menu (Figure 3.11), the primer is written to be understood by users familiar with simple secondary school Mathematics and no Computer Science background. It begins by introducing the DAC paradigm, using the eponymous Fibonacci sequence as a toy example, visualising its naive recursive implementation. Following this, readers are introduced to dynamic programming with a simple modification to the naive Fibonacci algorithm. Finally, the primer presents a high-level introduction to concept of algorithmic complexity and big-O notation, avoiding mathematical intricacies.

Throughout the primer we display visualisations of recursive call trees identical to the call trees seen in our visualisations, providing a gentle introduction to Divisualise interface.

3.7 Distribution

We aimed to make Divisualise widely accessible to a global audience and promote its adoption as a learning tool; to accomplish this, we used SvelteKit to build Divisualise as a static site that can be deployed on global content delivery networks (CDNs), allowing us to deliver the application to end users with minimal latency regardless of their location.

For the initial release of Divisualise, we chose to deploy the application on Vercel's Edge Network [Vercel, 2024], a global CDN that offers exceptional performance and reliability. Vercel's platform allows us to greatly simplify deployment by providing seamless CI/CD integrations with GitHub, and further allows us to host our application for free due to its small footprint!

In addition, we also made the decision to release the source code of Divisualise under an open-source license. By hosting the project on GitHub and making the code available under the GNU General Public License v3.0 (GPL-3.0), we aim to encourage collaboration and contribution from the wider computer science education community and hope to continue developing Divisualise into a comprehensive platform for learning about divide-and-conquer algorithms.

Chapter 4

API Design

In this chapter, we present the design of the Divisualise API, which aims to provide a simple, extensible, and elegant interface for creating divide-and-conquer algorithm visualisations. Our primary goal was to create an API that requires minimal effort from implementors while offering the flexibility to accommodate a wide range of algorithms. We wanted to ensure that teachers and advanced students could easily extend Divisualise to include new algorithms without needing to delve into the intricacies of the visualisation framework.

To achieve this, we designed the API around the core concept of modelling divide-and-conquer algorithms as recursive call trees, with each node representing a subproblem. By providing a set of abstract base classes and interfaces that encapsulate the common structure and behaviour of these call trees, we aimed to create a framework that allows implementors to focus on the specific details of their chosen algorithms while leveraging the power of the Divisualise platform. Crucially, we aimed to allow implementors to write only the naive recursive implementation of any algorithm, with Divisualise handling everything else, including input handling, presentation, and memoisation.

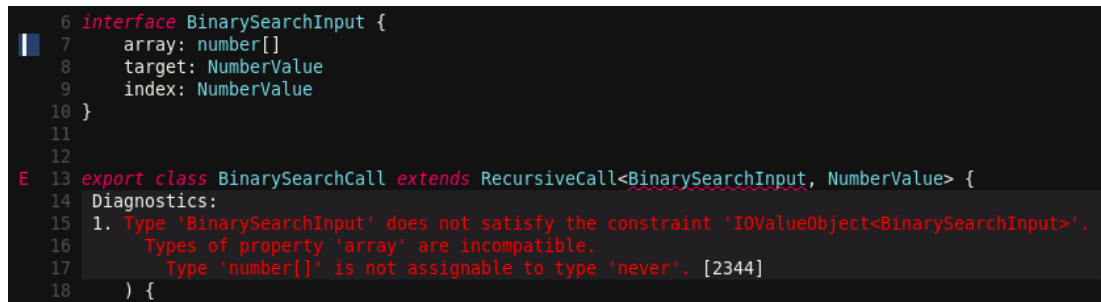
The following sections will detail the key components of the API, including the modelling of input and output values, recursive calls, and the configuration of new algorithms. Throughout this chapter, we present TypeScript code blocks to demonstrate key interfaces and methods. These figures are taken directly, with minimal modifications, from the source code, to showcase the elegance of the implementation that our abstractions allow (some redundant clauses needed to satisfy the type checker are removed for brevity and clarity).

4.1 Type Safety

One of our key objectives in designing the Divisualise API was to ensure type safety, as it provides several significant benefits for extensibility and ease of use by developers. By leveraging TypeScript's type system and support for algebraic data types (ADTs) [Liskov and Zilles, 1974], we were able to design an elegant API that catches potential errors at compile-time, reducing the likelihood of runtime bugs and making it easier for

teachers and advanced students to extend the platform.

Figure 4.1 demonstrates how the TypeScript compiler catches an input configuration error in the implementation of a binary search visualisation built using our API.



```

6 interface BinarySearchInput {
7   array: number[]
8   target: NumberValue
9   index: NumberValue
10 }
11
12
13 export class BinarySearchCall extends RecursiveCall<BinarySearchInput, NumberValue> {
14   Diagnostics:
15     1. Type 'BinarySearchInput' does not satisfy the constraint 'IOValueObject<BinarySearchInput>'.
16        Types of property 'array' are incompatible.
17        Type 'number[]' is not assignable to type 'never'. [2344]
18 }

```

Figure 4.1: TypeScript tells us that we cannot use the basic `number[]` type in the input of a `RecursiveCall` (see 4.2.1).

By embracing type safety, we aimed to create an API that is not only robust, but also more approachable for our target users. The compile-time type checking provides immediate feedback and guidance, making it easier for implementors to understand how to use the API correctly and reducing the friction associated with extending the platform to support new algorithms.

4.2 Modelling Input and Output Values

We began by considering how to handle input and output values. For our application, the basic types provided by JavaScript were insufficient as we wished to apply specialised styling to our values on the frontend; concretely, our frontend logic cannot differentiate matrices and planar points if we use the `number[][]` type to represent them both. Furthermore, we desired an API that supported arbitrary data types for ultimate extensibility.

To solve these issues, we defined wrapper classes over the basic data types which are augmented with additional presentational information and implement the `Value` interface (Figure 4.2). With this, `Divisualise` can work upon the abstract `Value` type and employ polymorphic dispatch at runtime to style values conditionally. Our `Value` interface ensures that items are copyable, and comparable for equality; these are necessary properties for memoisation, which we discuss further in subsection 4.3.3.

```

abstract class Value {
  abstract copy(): ThisType<this>;
  abstract copyDefault(): ThisType<this>;
  abstract equals(other: ThisType<this>): boolean;
}

```

Figure 4.2: The `Value` interface, implemented as an abstract class.

```

class NumberValue extends Value {
  value: number;
  colour: string = "black";
  struck: boolean = false;

  // constructor, copyDefault are trivial ...

  copy(): NumberValue {
    const copy = new NumberValue(this.value)
    copy.colour = this.colour;
    copy.struck = this.struck;
    return copy;
  }

  equals(other: NumberValue): boolean {
    return this.value === other.value;
  }

  coloured(colour: string): NumberValue {
    const copy = this.copy();
    copy.colour = colour;
    return copy;
  }

  struckThrough(struck: boolean): NumberValue {
    const copy = this.copy();
    copy.struck = struck;
    return copy;
  }
}

```

Figure 4.3: The (abridged) NumberValue class

Figure 4.3 presents the (partial) implementation of our NumberValue class, which tracks colour and strike-through styling information, alongside a simple numerical value. We used the copy method to provide an immutable API for styling numerical values; this immutable approach simplifies the implementation of details on recursive calls - this is discussed further in subsection 4.3.5.

We implemented NumberList, Matrix, and Points classes similarly, allowing us to flexibly present a range of common data types within our visualisations. Our NumberList and Matrix classes leverage our NumberValue class to provide styling capabilities. Our Points class is built independently and enables further functionality for colouring points, and drawing arbitrary straight-line segments on the plane to serve as visual aids; to accomplish this, we created two further subclasses of Value (Point and Line) and, as a result, had to define a new type which narrows to the Value types we wished Divisualise to consider (Figure 4.4).

```

type IOValue = NumberValue | NumberList | Matrix | Points

```

Figure 4.4: The IOValue type

With this, extending Divisualise to work with arbitrary new types becomes trivial! One must simply: define a subclass of `Value`; extend `IOValue` to accept this new type; and build two Svelte components which handle input, and presentation, of the type (see section 5.1). This approach greatly simplifies the process of adding support for new data types, making it easy for implementors to extend Divisualise to visualise algorithms that operate on custom data structures.

4.2.1 Sets of Named Values

We desired a mechanism to label input and output values for display in Divisualise. Furthermore, we needed a way to handle multiple input/output variables to enable the visualisation of certain algorithms; for instance: Binary Search necessitates an input array, as well as a target value to find. To handle arbitrary, multivariate recursive functions in a type-safe manner, we defined one more type:

```
type IOValueObject<T> = {
  [K in keyof T]: T[K] extends IOValue ? T[K] : never;
};
```

Figure 4.5: The `IOValueObject` type

With this, we represent a set of named values as plain JavaScript objects, with entries restricted to the `IOValue` type. An object which satisfies `IOValueObject` can be viewed as an ordered (FIFO when iterating over entries - a useful property for ordering content for the frontend) key-value store, with string keys and `IOValue` reference values. This design choice enables implementors to easily define the input and output types for their algorithms, while ensuring type safety and providing a consistent structure for the Divisualise frontend to work with.

4.3 Modelling Recursive Calls

We model divide-and-conquer algorithms as trees of subproblems, where each node in the tree represents a recursive call. To capture this structure, we define a data model representing a recursive call as having distinct 'divide' and 'combine' phases, executed independently.

The following subsections will detail how we model the state of a recursive call (subsection 4.3.1), define the recursive call and recursive case classes (subsection ??), and enable powerful features such as memoisation (subsection 4.3.3) and interactive exploration of the call tree (subsection 4.3.4).

4.3.1 Modelling the State of a Recursive Call

We modelled the state of a recursive call as a sum type, implemented in TypeScript as a discriminated union of interfaces, with four possible variants:

UndividedState represents a call that has not yet been divided into subproblems. It has no result or case, but may contain a memoisedPath property in a memoised call tree.

DividedState indicates that the call has been divided into subproblems, which are stored in a case property.

SolvedState signifies that all subproblems in its case have been solved, and their results have been combined to produce the final result for the call, which is stored in the result property.

MemoisedState is only applicable in a memoised call tree; this is a special case that represents a call whose input has been encountered, and *necessarily evaluated*, earlier in the call tree. When a call is memoised, it stores the 'path' (see subsection 4.3.3) to the first occurrence of the subproblem from the root node in the memoisedPath property, allowing the memoised call and its result to be retrieved without recomputation. The memoised state is transitioned to directly from the UndividedState, and, crucially, introduces an order-dependence in the execution of the call tree.

Additional characteristics of a recursive call, such as whether it is 'combinable', or is a base case, can be derived with the state of its subcalls. For instance: we say a call is combinable if it is of DividedState, and all its subcalls are of SolvedState - notice this holds vacuously for base cases with zero subcalls, allowing for elegant implementation of further functionality, as detailed in subsection 4.3.4.

4.3.2 Recursive Calls and Recursive Cases

We defined two generic abstract classes—RecursiveCall<In, Out> and RecursiveCase<In, Out>—where In and Out are type variables which necessarily satisfy IO-ValueObject.

```
abstract class RecursiveCall<
  In extends IOValueObject<In>,
  Out extends IOValue | IOValueObject<Out>
> {
  input: In
  state: RecursiveCallState<In, Out>
  root: RecursiveCall<In, Out>
  memoisedResults: MemoisedResult<In, Out>[] = []
  memoise: boolean = false

  abstract case(input: In, root: RecursiveCall<In, Out>):
    RecursiveCase<In, Out>
}
```

Figure 4.6: The (abridged) RecursiveCall abstract class

Figure 4.6 presents the definition of the RecursiveCall class; the input property is of type In, the state property is a discriminated union type (as discussed in subsection

4.3.1), and the root property references the root node of the call tree—a design decision which enables the memoisation functionality detailed in subsection 4.3.3.

The RecursiveCase class is further specialised into two subclasses: DivideCase and BaseCase, as shown in Figure 4.7.

```
abstract class DivideCase<
  In extends IOValueObject<In>,
  Out extends IOValue | IOValueObject<Out>
> extends RecursiveCase<In, Out> {

  abstract divide(input: In, root: RecursiveCall<In, Out>):
    RecursiveCalls<In, Out>

  abstract combine(): Out
}

abstract class BaseCase<...> ... {
  abstract solve(input: In): Out
}
```

Figure 4.7: The (abridged) DivideCase and BaseCase abstract classes

The DivideCase class represents a recursive case where the problem is divided into subproblems; it defines two abstract methods: divide, which splits the input into subproblems, and combine, which merges the results of the subproblems. The BaseCase class represents a base case where the problem is solved directly, without further recursion; it defines a single abstract method, solve, which computes the solution for the base case input.

The divide method of DivideCase returns a value of type RecursiveCalls<In, Out>, which is a helper type defined as Record<string, RecursiveCall<In, Out>. This design choice mirrors the IOValueObject type, providing a consistent way to represent a set of named recursive calls, similar to how IOValueObject represents an ordered set of named input/output values. By using a record type with string keys, we enable the creation of visualisations for divide-and-conquer algorithms with arbitrary branching factors, enhancing the flexibility and extensibility of our API.

Crucially, by defining the recursive call and recursive case classes in this manner, we allow implementors to focus solely on writing the naive recursive logic for their chosen algorithms. The implementor need only override the divide and combine methods for divide cases, and the solve method for base cases, with Divisualise handling all other aspects of the visualisation, including input handling, presentation, and interaction. This design greatly simplifies the process of extending Divisualise to support new algorithms, as the implementor is not required to understand the intricacies of the visualisation framework.

4.3.3 Memoisation for Free

Memoisation is a powerful optimisation technique that can significantly improve the performance of certain recursive algorithms by avoiding redundant computations. The RecursiveCall API provides built-in support for effortless memoisation, allowing users to seamlessly switch between naive recursive and memoised versions of an algorithm. Implementors can enable memoisation for their algorithms by simply overriding the `isMemoisable` method to return `true`; Divisualise will then handle the memoisation process automatically, building upon the naive recursive implementation provided by the implementor.

To enable memoisation, the RecursiveCall constructor accepts a reference to the root node of the call tree. This allows us to identify each call by its 'path' from the root, which serves as a unique identifier for the subproblem. We represent the path of a call as a list of subcall label strings encountered when traversing from the call's root to itself. For example, the root node of any tree has path `[]` - the empty list; the first non-root evaluated node in a merge sort tree has path `["left"]`.

When constructing the root node of a call tree, if the algorithm is memoisable, the constructor sequentially evaluates an identical call tree in order, avoiding redundant computations, to determine the paths at which the first occurrences of each subproblem are solved. This information is stored in the `memoisedResults` property of the root node. Figure 4.8 shows the section of the RecursiveCall constructor which implements this logic.

```
if (this.isRoot() && this.isMemoisable()) {
  const shadowCall = new this.constructor(copy(input), ...)
  shadowCall.root = shadowCall
  shadowCall.memoise = true
  while (!shadowCall.isSolved()) {
    const traversed = shadowCall.step()
    if (traversed.isStrictlySolved()) {
      shadowCall.memoisedResults.push({
        input: copy(traversed.input()),
        result: copy(traversed.result()),
        firstSolvedPath: shadowCall.pathOf(traversed)
      })
    }
  }
  this.memoisedResults = shadowCall.memoisedResults
}
```

Figure 4.8: Finding first subproblem occurrences in the RecursiveCall constructor.

By tracking the first solved paths for each subproblem, we can maintain a consistent state in the presence of memoisation, even when the user explores the call tree in an arbitrary order. More information on how we accomplished this is presented in the next subsection, 4.3.4.

4.3.4 Interacting with the Call Tree

With our abstractions cemented, we proceeded to write functionality atop our abstract methods to enable intricate interaction with, and exploration of, our recursive call trees. This subsection presents implementations of the main methods of interest in the RecursiveCall API - each method corresponds directly to an interaction seen on the Divisualise interface.

Firstly, we present the `divide()` method in Figure 4.9. When a call is divided, it checks whether its input is encountered at an earlier path in the tree, and if a call exists at that (possibly null) path. If an earlier call exists, the call in consideration transitions directly to the memoised state, bypassing the need to recompute the subproblem; otherwise, the next case is obtained by calling the abstract case method, and the state of the call becomes `DividedState` accordingly.

The `combine` method is comparatively uncomplicated - we simply check that the call is 'combinable' (as defined in 4.3.1), and update the state to `SolvedState` accordingly.

```
divide() {
  if (!this.isDivisible()) {
    return
  }
  const memoised = this.getMemoisedResults()
    .find(mem => equal(mem.input, this.input))
  if (
    this.isRootMemoised()
    && memoised !== undefined
    && !pathsAreEqual(this.pathFromRoot(), memoised.
      firstSolvedPath)
  ) {
    this.state = {
      type: "memoised",
      memoisedPath: memoised.firstSolvedPath,
      result: memoised.result
    }
  }
  else {
    const nextCase = this.case(copy(this.input, this.root)
    this.state = {
      type: "divided",
      case: nextCase
    }
  }
}
```

Figure 4.9: The RecursiveCall `divide` method.

Next, we present the `step` and `conquer` methods in Figure 4.10. The `step` method performs a single step towards solving the call; it recursively traverses the call tree, in order, to find the next (sub)call to divide or combine, then return. If called on a solved call, `step` returns null to indicate that no call was acted upon. The `conquer` method is trivial, repeatedly invoking `step` until the call is solved.

```

step(): RecursiveCall<In, Out> | null {
    if (this.isDivisible()) {
        this.divide()
        return this
    }
    else if (this.isDivided()) {
        for (const call of this.subcalls()) {
            if (!call.isSolved()) {
                return call.step()
            }
        }
        this.combine()
        return this
    }
    else {
        return null
    }
}

conquer() {
    while (!this.isSolved()) {
        this.step()
    }
}

```

Figure 4.10: The step and conquer methods.

Finally, we discuss the more complex `reset` and `toggleMemoise` methods. Due to the size and intricacy of these methods, we provide only high-level overviews of their logic and direct interested readers to the source code.

The `reset` implementation resets a call to `UndividedState` - at a glance, this appears to be a trivial operation, however it requires careful attention. Transitioning a call from `SolvedState` to `UndividedState` can change whether or not its parent call is 'combinable', leading to inconsistent state if the parent is already solved! To ensure our tree always maintains consistent state, we designed an algorithm that operates as follows:

1. Unsolve (return to `DividedState`) all solved calls on the path to the reset call from its root, keeping track of all unsolved call paths in a set of 'affected paths'.
2. Add the paths of all subcalls of the reset call to the set of affected paths.
3. If we're operating in a memoised call tree, traverse the tree from the root, in reverse order, and recursively call `reset` on any memoised calls with a `memoisedPath` equal to any of our affected paths.
4. Change the state of the reset call to `UndividedState`.

The `toggleMemoise` method maintains consistent state as memoisation is enabled and disabled on a recursive call tree; it can only be called on the root node of a memoisable recursive call tree and functions as follows:

1. Set the memoise flag to true if it is false, and vice versa.
2. If we have just turned memoisation off, we traverse the tree find all memoised calls, unsolve them, and conquer them, expanding their full subproblem tree.
3. Otherwise, we have turned memoisation on and the process is more challenging. We traverse all calls in our tree in reverse; if we encounter a call for which we have a memoised result (subsection 4.3.3), we carefully adjust its state to ensure consistency (again, we point the reader to the source code, as the logic in this step is particularly intricate).

4.3.5 Providing Details

To enable the creation of detailed, animated explanations of the state of each recursive call, we began by modelling our animated slideshows with the `CallDetails` type shown in Figure 4.11.

```
interface CallDetailsStep<> {
    text: string
    valueKeyframes?: Record<string, IOValue>[]
    highlightedCalls?: string[]
}

type CallDetails = CallDetailsStep<any>[]
```

Figure 4.11: The `CallDetails` type.

Here, an object which fits the `CallDetailsStep` interface can be seen as one slide in our slideshow. Each slide contains descriptive text, an optional list of key-frames which animate stylised values within the slide, and an optional list of subcall labels to be highlighted.

Following this, we defined further abstract methods—`undividedDetails` on `RecursiveCall`, `dividedDetails` and `undividedDetails` on `RecursiveCase`—of return type `CallDetails`. With these new methods, we could define the concrete details method on `RecursiveCall`, which conditionally dispatches to our abstract methods to return `CallDetails` depending on the call’s state.

Our approach enables the procedural generation of call detail slideshows as they are needed at run-time, avoiding unnecessary computation and improving the performance of the app. Figure 4.12 shows how the interface we defined is used to create a slideshow for a base case in the Fibonacci algorithm.

```

solvedDetails(input: FibonacciInput): CallDetails {
    return [{
        text: "We return 1.",
        valueKeyframes: [{
            "Result": new NumberValue(1)
        }]
    }]
}

```

Figure 4.12: The FibonacciBaseCase solvedDetails method.

4.4 Configuring New Algorithms

To facilitate the seamless integration of new algorithms into Divisualise, we designed the AlgorithmConfig interface, which encapsulates all the necessary information for configuring an algorithm for use on the frontend. Figure 4.13 presents the definition of this interface.

```

type InputType = "Number" | "NumberList" | ...

type IconType = "Matrix" | "Sort" | ...

interface AlgorithmConfig<
    In extends IOValueObject<In>,
    Out extends IOValue | IOValueObject<Out>
> {
    name: string
    icon: IconType
    callConstructor: typeof RecursiveCall<In, Out>
    inputs: Record<string, InputType>
}

```

Figure 4.13: The AlgorithmConfig interface.

The name field specifies the display name of the algorithm, while the icon field determines the icon used to represent the algorithm in the user interface. The callConstructor field takes the constructor of the concrete RecursiveCall subclass for the algorithm; this allows Divisualise to dynamically instantiate the appropriate call tree for the selected algorithm. We defined the inputs field as a record that maps input variable names to their corresponding InputType, allowing Divisualise to automatically handle input for each algorithm (see 5.1).

By encapsulating the configuration of an algorithm in the AlgorithmConfig interface, we provide a clear and type-safe way to integrate new algorithms into Divisualise. Implementors need only define the necessary types and provide the required information in the config object; Divisualise takes care of the rest, dynamically generating the user interface and input components based on the provided configuration. This approach promotes extensibility and maintainability, making it easy for teachers and advanced students to add new algorithms to the visualiser.

Chapter 5

Frontend Implementation

In this chapter, we very briefly discuss the key design decisions and techniques employed in the frontend implementation of Divisualise, focusing on creating an intuitive, responsive, and accessible user interface. The Divisualise frontend builds upon the RecursiveCall API (chapter 4), using SvelteKit and Tailwind CSS to create beautiful, engaging, and interactive algorithm visualisations.

5.1 Input and Output Values

The dynamic rendering of input and output values is essential for visualising a wide range of algorithms and data structures in Divisualise. Our Input Svelte component accepts an object satisfying AlgorithmConfig (see 4.4) as a 'prop' (essentially, a component parameter) and dynamically dispatches to type-specific input components, such as NumberInput and PointsInput, based on the input types specified in the configuration object. This approach allows us to support arbitrary input types while maintaining a modular structure that is easy to extend.

We took an identical approach to displaying arbitrary output values, with one main Value component dispatching to specialised components such as NumberValue and Matrix. Our output components utilise the styling information provided by the Value subclasses to dynamically style presented values.

To ensure responsiveness across different screen sizes, we used Tailwind CSS responsive breakpoints to dynamically resize and adapt our input and output components with the amount of screen real-estate available.

5.2 The RecursiveCall Component

The RecursiveCall Svelte component serves as the visual representation of a node in an algorithm's call tree. In developing the RecursiveCall component, we used a variety of nifty Svelte and CSS tricks to display beautiful, interactive call trees.

The RecursiveCall component integrates tightly with the RecursiveCall API. Each

rendering of the `RecursiveCall` component tracks an instance of an object which implements the `RecursiveCall` interface (4.3), and uses the `RecursiveCall` API to provide interactivity. Further, our component utilises the predicate methods exposed by the `RecursiveCall` API to conditionally display state information and interactive elements, ensuring that the component accurately reflects the current state of the call tree.

The most interesting aspect of the the `RecursiveCall` component design is the way it recursively renders itself to display a complete, interactive call tree. By using the lesser-known recursive features of Svelte components, we render a call tree using dynamic HTML and CSS, rather than relying on canvas, SVG, or other rendering APIs. This choice ensures that the call tree can be fully navigated and understood using a keyboard or assistive technologies, as the structure and meaning of the call tree are preserved in the DOM, greatly improving the accessibility of our visualisations.

We made extensive use of the CSS Flexbox API to allow our call tree nodes to resize themselves based on the size of their input and output values, and to allow our entire call tree to dynamically resize itself as its branches are explored and reset. To draw branches between our tree nodes we also used Flexbox, alongside Svelte conditional rendering, to draw dynamically-sized invisible containers with partial black borders that connect a recursive call to its children.

5.3 The Divisualise Component

The `Divisualise` component serves as the top-level container for an entire algorithm visualisation, bringing together the call tree, details panel, and playback controls into a cohesive user interface.

We placed our visualisation playback logic within the `Divisualise` component; this logic works in a hierarchical fashion, stepping over the recursive call tree, detail slides, and slide key-frames. Crucially, our playback logic takes into account the current state of the visualisation, the currently highlighted call, and the status of the details panel, to provide a smooth and adaptive playback experience. Given the number of moving parts, our playback logic is particularly intricate; we used the asynchronous programming features present in JavaScript to provide structure in its implementation.

Another noteworthy behavior of the `Divisualise` component is the way it automatically centres the camera on the currently highlighted call. It accomplishes this by dynamically calculating the screen-space bounding box of calls as they are highlighted, transforming their coordinates into camera-space, and smoothly panning the camera to these coordinates.

Again, to enhance usability and accessibility, we used the responsive design capabilities of the Tailwind CSS framework. By employing a mobile-first approach and utilising responsive utility classes, we have created a layout that adapts seamlessly to different screen sizes and orientations. The responsive design extends to the playback controls and other interactive elements, ensuring that they remain easily accessible and usable across a wide range of devices.

Chapter 6

User Evaluation

To assess whether or not Divisualise had met its design goals, and to identify potential areas for improvement, we conducted an anonymised user survey, which we placed on the landing page of Divisualise after making the first version of the application publicly available. We opted for an anonymised survey due to the fact that the first version of Divisualise was publicised through the author's personal social network and the University of Edinburgh's School of Informatics undergraduate mailing list. As many of the potential respondents were likely to know the author personally, we wanted to ensure they felt comfortable providing honest feedback without the concern that their responses could be traced back to them.

We considered designing a study to test the effectiveness of Divisualise in improving learning outcomes, but ultimately decided against it. Designing such an experiment would require considerable effort and necessitate a large sample size to demonstrate a significant result, which we deemed to be beyond the scope of this project. Instead, we focused on gathering user feedback through the survey to gain insights into the usability, accessibility, and perceived value of Divisualise.

6.1 Survey Design

The survey collected both quantitative and qualitative data from users, allowing us to assess the extent to which we have met our design goals and to identify potential areas for improvement. We began by collecting background information about the respondents, such as their level of computer science education, experience as software engineers, and experience teaching computer science. This information was intended to help us contextualize the responses and identify any trends based on the respondents' prior knowledge and experience.

The body of the survey presented a series of questions designed covering topics such as ease of use, intuitive interface design, aesthetic appeal, and effectiveness in improving understanding of divide-and-conquer algorithms. We used a mixture of Likert scales and binary agree/disagree statements to allow us to quantify the extent to which users agreed with the presented statements. In addition, we also included open-ended questions

to collect qualitative feedback from users; these questions provided respondents with the opportunity to share their thoughts on how Divisualise could be improved, and to provide any additional feedback they felt was relevant. By including these open-ended questions, we aimed to gather valuable insights that could inform future development and extensions of the application.

6.2 Quantitative Insights

In total, we collected responses from 28 users with varied computer science backgrounds, ranging from no experience whatsoever to professional computer educators. The responses to our survey provide strong evidence that Divisualise has met its design objectives.

The majority of respondents found the application easy to use, with 89% rating the ease of use as a 4 or 5 on a 5-point scale. Similarly, 93% of respondents agreed that the user interface is intuitive and Divisualise feels nice to use. In terms of aesthetic appeal, 93% of respondents rated the Divisualise interface as a 4 or 5 on a 5-point scale, suggesting that we have succeeded in creating a visually pleasing and engaging user experience. Furthermore, 86% of respondents rated the overall design of the interface as a 4 or 5.

Regarding the educational value of Divisualise, all respondents indicated that using the application had improved their understanding of DAC algorithms (or of one particular DAC algorithm) to some extent, with 57% reporting a significant improvement (4 or 5 on a 5-point scale). Additionally, 96% of respondents believed that Divisualise could be significantly valuable to computer science educators teaching divide-and-conquer algorithms, and 86% agreed that they would want to use Divisualise if they were learning about DAC algorithms for the first time.

96% of respondents agreed that Divisualise accurately demonstrates the workings of DAC algorithms, and 86% of respondents agreed that Divisualise covers a good range of divide-and-conquer algorithms.

6.3 Qualitative Feedback

While responses were largely positive, respondents also provided valuable feedback on areas where the application could be improved. One common criticism was the lack of an obvious way to return to the home page or select a different algorithm after visualising one, with several respondents mentioning that they had to refresh the page to try a different algorithm or enter new data. This design oversight has since been addressed with the addition of a clear 'home' button. Respondents also suggested that providing more detailed explanations or descriptions of the algorithms and their steps would be helpful, particularly for users with a limited understanding of divide-and-conquer algorithms. Other suggestions for improvement included allowing users to easily generate random mock data for testing algorithms, providing a colour scheme chooser to enhance customisation, and displaying equations or formulas used in certain algorithms to clarify the underlying mathematics.

Chapter 7

Conclusions

In this project, we set out to create Divisualise, an extensible, interactive divide-and-conquer algorithm visualiser.

Our primary objectives were to develop a tool that is intuitive, informative, and accessible to users with varying levels of expertise in computer science, and to create a tool which can be used effectively by teachers in a classroom setting to demonstrate the DAC paradigm. To achieve these goals, we designed Divisualise around the idea of modelling DAC algorithms as interactive recursive call trees. We created a simple, yet powerful, API that allows implementors to write naive recursive implementations of DAC algorithms, while allowing the platform to handle the intricacies of input handling, presentation, and optimisations such as memoisation.

To evaluate the effectiveness of Divisualise in meeting our objectives, we conducted a user survey that provided valuable insights into the strengths and weaknesses of the platform. The survey results demonstrate that Divisualise has largely succeeded in its mission, with the vast majority of respondents finding the application easy to use, intuitive, and aesthetically pleasing. Furthermore, a significant proportion of users reported an improved understanding of divide-and-conquer algorithms after using Divisualise, and believed that the platform could be a tool of significant value for computer science educators.

While the survey results are positive, they highlight several areas where Divisualise can be improved. We intend to address these criticisms and suggestions in future iterations of the platform, with the goal of creating an even more comprehensive and user-friendly tool for learning and teaching divide-and-conquer algorithms. One of our primary aims for the future of Divisualise is to foster an open-source community around the project, transforming it into a one-stop-shop platform for learning about divide-and-conquer algorithms.

Some potential avenues for future development include:

1. Adding support for LaTeX in presentational text, enabling better mathematical notation in presentations and explanations.
2. Generalising the Divisualise approach to create interactive visualisations of itera-

tive algorithms, potentially establishing a new state-of-the-art general algorithm visualiser.

3. Incorporating detailed write-ups about specific divide-and-conquer algorithms to cater to the needs of both novice and advanced students.
4. Extending the `CallDetails` interface to support additional content types, such as code snippets, to provide a more comprehensive understanding of each step in an algorithm's execution.
5. Implementing a wider range of data types and algorithms, further showcasing the flexibility and extensibility of the Divisualise platform.

In conclusion, Divisualise represents a step forward in the realm of algorithm visualisation, improving on its predecessors by demonstrating high-level algorithmic structure and low-level details in tandem, and offering a powerful and intuitive tool for learning and teaching DAC algorithms. By combining an elegant and extensible API with a user-friendly and accessible frontend, Divisualise has the potential to grow into a comprehensive platform for learning about divide-and-conquer algorithms. As we continue to refine and expand the platform, we are confident that Divisualise will become an indispensable resource for anyone seeking to master the DAC paradigm.

Bibliography

- Adream Blair-Early and Mike Zender. User interface design principles for interaction design. *Design Issues*, 24(3):85–107, 2008. ISSN 07479360, 15314790. URL <http://www.jstor.org/stable/25224185>.
- Marc H. Brown and Robert Sedgewick. A system for algorithm animation. *SIG-GRAPH Comput. Graph.*, 18(3):177–186, jan 1984. ISSN 0097-8930. doi: 10.1145/964965.808596. URL <https://doi.org/10.1145/964965.808596>.
- Thomas H. Cormen et al. *Introduction to Algorithms*. Third edition, 2009.
- Department for Education. Technology in schools survey report: 2022 to 2023, 2023. URL, <https://www.gov.uk/government/publications/technology-in-schools-survey-report-2022-to-2023> [Accessed: March 2024].
- Eric Fouh et al. The role of visualisation in computer science education. *Computers in the Schools*, 2012.
- Google. Mobile web traffic statistics, 2024. URL, <https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/mobile-web-traffic-statistics/> [Accessed: March 2024].
- Christopher Hundhausen, Sarah Douglas, and John Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages Computing*, 13:259–290, 06 2002. doi: 10.1006/jvlc.2002.0237.
- Barbara Liskov and Stephen Zilles. Programming with abstract data types. *SIGPLAN Not.*, 9(4):50–59, mar 1974. ISSN 0362-1340. doi: 10.1145/942572.807045. URL <https://doi.org/10.1145/942572.807045>.
- Thomas Naps, Guido Rößling, Vicki Almstrum, Wanda Dann, Rudolf Fleischer, Christopher Hundhausen, Ari Korhonen, Lauri Malmi, Myles McNally, Susan Rodger, and J. Ángel Velázquez-Iturbide. Exploring the role of visualization and engagement in computer science education. *SIGCSE Bulletin*, 35:131–152, 05 2003.
- Svelte contributors. SvelteKit, 2024. URL <https://kit.svelte.dev/> [Accessed: March 2024].
- Tailwind CSS contributors. Tailwind CSS, 2024. URL, <https://tailwindcss.com/> [Accessed: March 2024].
- N Tractinsky, A.S Katz, and D Ikar. What is beautiful is usable.

- Interacting with Computers*, 13(2):127–145, 2000. ISSN 0953-5438. doi: [https://doi.org/10.1016/S0953-5438\(00\)00031-X](https://doi.org/10.1016/S0953-5438(00)00031-X). URL <https://www.sciencedirect.com/science/article/pii/S095354380000031X>.
- Vercel. Edge Network, 2024. URL <https://vercel.com/docs/edge-network/overview> [Accessed: March 2024].