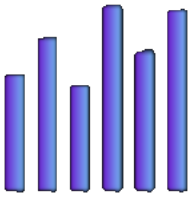
CODE VIZ



Karun Mahadevan

App Overview:

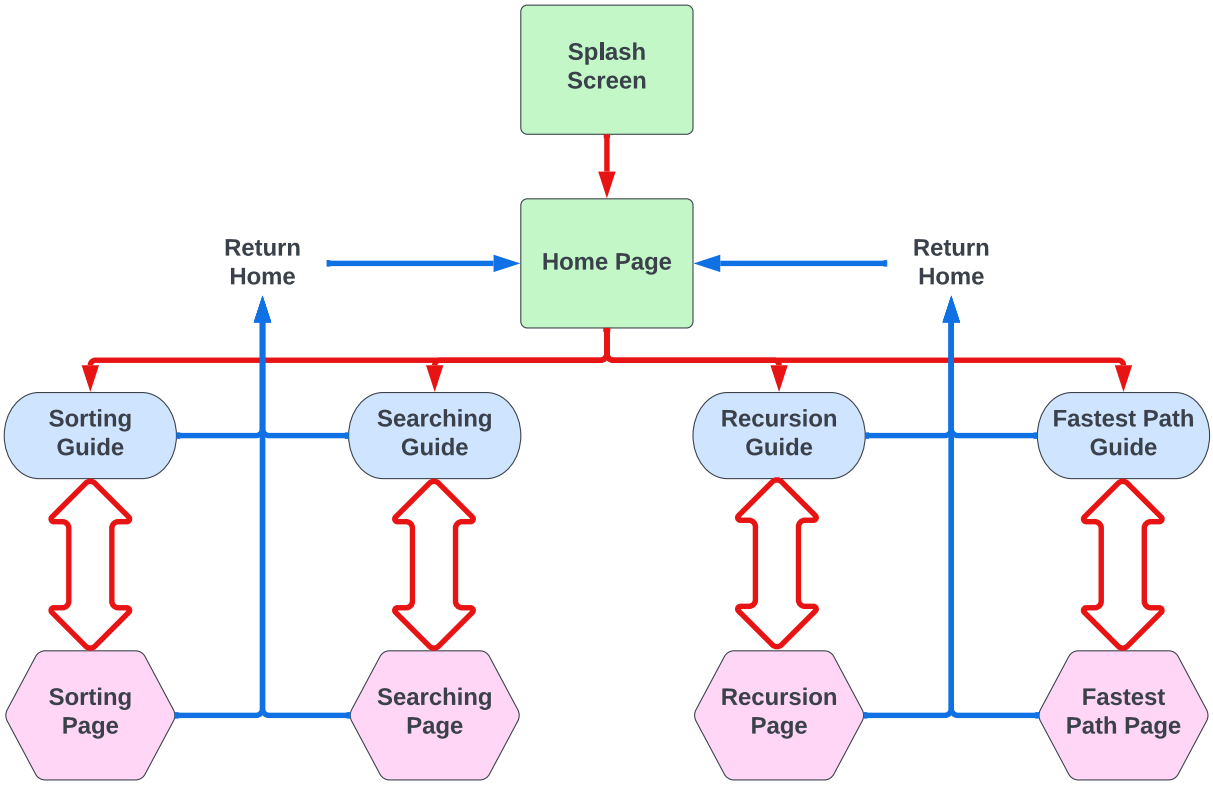
CodeViz is an all-in-one code visualization tool. There are apps that visualize sorting *or* searching, but not both. There is only one app that shows any other type of algorithm, called "Algorithm Visualizer"; the reviews for this app are negative, noting a poor UI and poor visualization. The premise of CodeViz is to elegantly display essential sorting and searching algorithms, as well as backtracking-recursion and depth-first search. The following are the visualized algorithms:

1. Sorting
    a. Bubble Sort
    b. Selection Sort
    c. Insertion Sort
    d. Quick Sort
    e. Merge Sort
2. Searching
    a. Linear Search
    b. Optimized Linear Search
    c. Binary Search
    d. Ternary Search
    e. Jump Search
    f. Exponential Search
3. Recursion
    a. Backtracking
4. Breadth-First Search
    a. Fastest Path

The two significant features leveraged in CodeViz are the MPAndroid Chart library, and Kotlinx Coroutines (referred to as Coroutines for the rest of the report). MPAndroid Chart is an extremely powerful charting library which affords the programmer extreme control on composition of graphs. The use of this expansive library was vital in dynamically constructed bar charts that are used for visualizing sorting and searching. Coroutines are a Kotlin-native tool for concurrent programming, that simplify writing asynchronous code to ensure main-safety as well as protection for long-running tasks. Due to the CPU-intensive work of sorting and searching, utilizing Coroutines was essential to successfully creating CodeViz.

User Experience:

The user experience flow for CodeViz is detailed in this flow chart:



- The Splash screen (fig. 1) automatically leads to the home page (fig. 2).
- From here, the user can select any of the activities to visualize, which lead to the guide pages (fig. 3). The guide pages are scroll views that provide a tutorial on how to use the following visualizer.
- Each guide page provides the option to return to the home page (blue arrows) or proceed to the visualizing page (hollow red arrow).
- Each visualization page has the capability of returning to the home page (blue arrows) or going back to the guide page (hollow red arrow).

The following details the user experience for each visualization page:

1) **Sorting** (figures 4 and 5)
   a) A material slider lets the user select the number of values that will be sorted (fig. 4).
   b) A spinner lets the user select which of the 5 sorting methods are visualized (fig. 4).
   c) When the user clicks “sort chart”, the selected sort method will be visualized (fig. 5).
2) **Searching** (figures 6 and 7)
   a) A material slider lets the user select which value will be searched for (fig. 6).
   b) A spinner lets the user select which of the 6 searching methods are visualized (fig. 6).
   c) When the user clicks “search through data” the selected search method will be visualized (fig. 7).
3) **Recursion** (figures 8 and 9)
   a) A material slider lets the user change the size of a “chess board” which will visualize the “N-Queens” problem (fig. 8).
   b) When the user clicks “solve n-queens”, the N-Queens problem of a selected size will be solved, visualizing the recursive backtracking solution (fig. 9).
4) **Fastest Path**
   a) A custom view lets the user “draw” which points will be blocked in the fastest path from green square to red square (fig. 10).
   b) The drawing on the custom view is transformed into a 25x25 grid with the drawing superimposed as black circles.
   c) The breadth first search proceeds, with purple indicating either a completed path or an unexplored point, red indicating the point is currently being searched, or white indicating a previously searched grid position (fig. 11).
   d) When the search is finished, the user can click a button to superimpose the fastest path (fig. 12).

Implementation:

CodeViz utilized a multitude of features that required an independent learning process, such as fragments (for each visualization), scroll views (for guide pages), and shaders (for custom text coloring). While those are all valuable, the most important features that CodeViz needed were MPAndroid Chart and Coroutines.

The use of MPAndroid Chart is a major reason why CodeViz is a successful application. MPAndroid chart offers a plethora of customization, which allowed me to construct the graph exactly how I wanted to. Selecting which grid and axis lines to show, and when to show values or hide them were all conscious decisions I made to enhance the user's experience. As seen in the splash screen animation (fig. 1, animation not shown), MPAndroid's inherent animation ability allows for a complex UI feature with minimal implementation work. Besides the customizability, by leveraging the inherent structure of an MPAndroid Bar Chart I was able to significantly minimize program overhead while still visualizing extremely intensive processes.

Since a MPAndroid Bar Chart's data is held in an array list of Bar Entries (the X and Y values of a bar in the chart), I represented the Y values in their own array, and would use a loop to fill the Bar Entry array with the index, and corresponding Y values. Because of this, I was able to sort and search through my Y values array, and directly update every bar at the same time, rather than having to reconstruct my entire chart for each iteration. The same principle is applied to coloring the individual bars of the array. Because effective visualization was paramount for this app, being able to represent the bar colors in a simple array list was extremely valuable. When sorting/searching through the Y values array, I would perform the necessary action in the bar color array, leveraging all the tools MPAndroid Chart has to offer.

Along with MPAndroid Chart, CodeViz would not have been possible without Coroutines. The Kotlin-native concurrent programming feature allowed me to ensure main-safety (not overworking the main thread) for asynchronous tasks. Because all the algorithms visualized are very computationally expensive, utilizing the different scopes that Coroutines offers let me ensure that all processes ran smoothly. Coroutines offer three distinct scopes – Main (the main thread, for interacting with the UI), IO (for input and output, network requests), or Default (multi-threaded, for CPU intensive work).

In CodeViz, I utilize the Default Scope to run all algorithms, then when necessary to update the UI for visualization the Main Scope is used. The beauty of Coroutines is that stopping and resuming of asynchronous tasks is automated so when switching to the Main Scope is called, the multi-thread management of the Default Scope is automatically handled and paused. As all the algorithms are called, the Default Scope is launched. Then, for each iteration/recursive call, the Main Scope suspends the Default Scope, updates the UI as needed, then resumes the Default Scope.

Miscellaneous:

The motivation for making this app is the fact that I am a very visual learner. Looking at raw code for these algorithms often leads me to more confusion; I understand the code but don't get *why* the code works. While making this app, I was learning as visualizations were completed, and the cyclical learning process not only let me enhance the app but solidified my understand of these essential algorithms.

Appendix: Photos

Figure 1

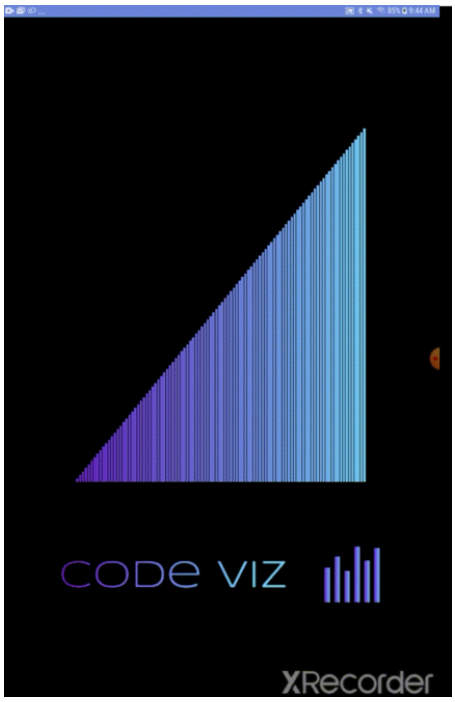


Figure 2

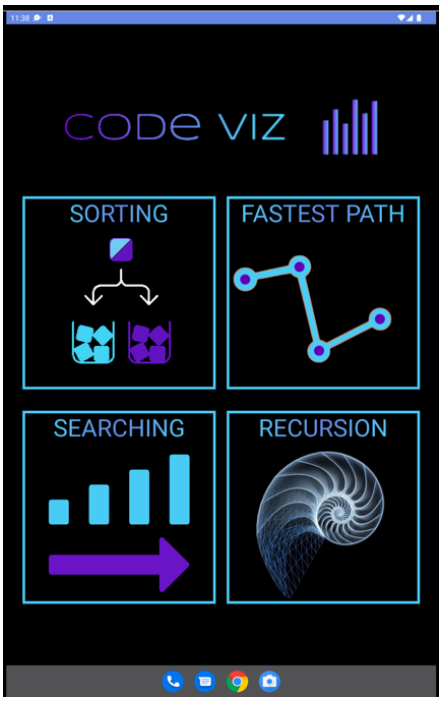


Figure 3

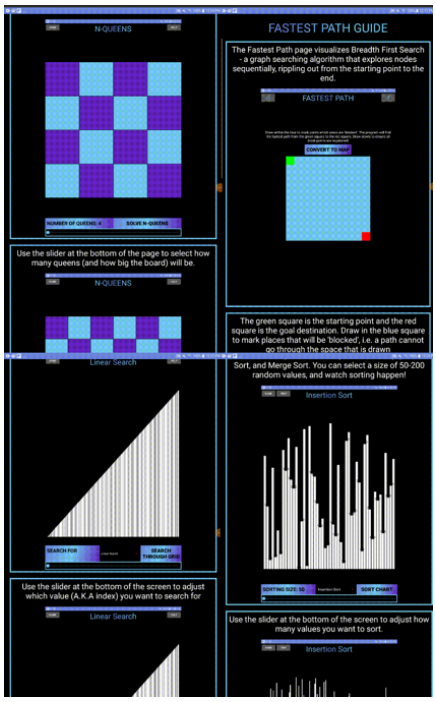


Figure 4

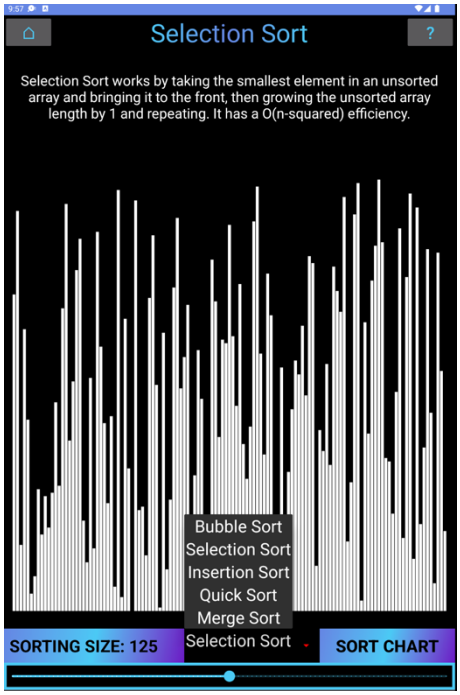


Figure 5

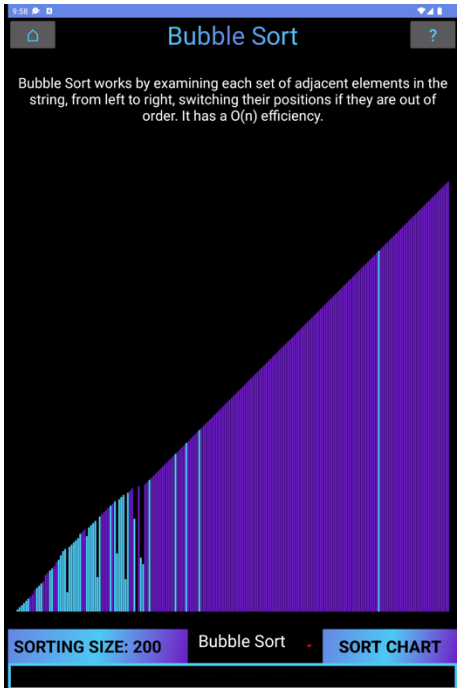


Figure 6

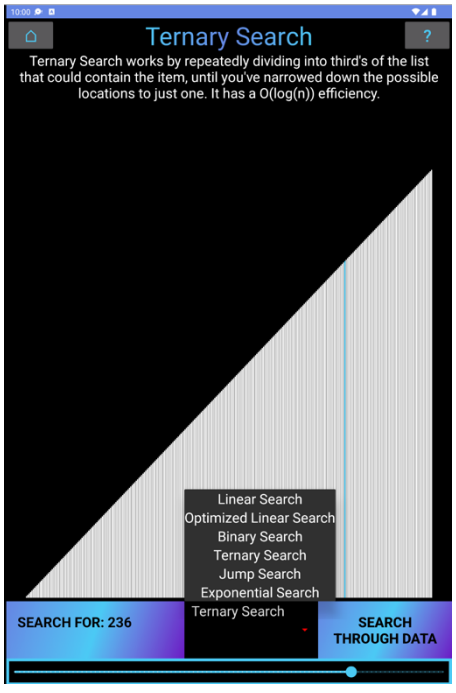


Figure 7

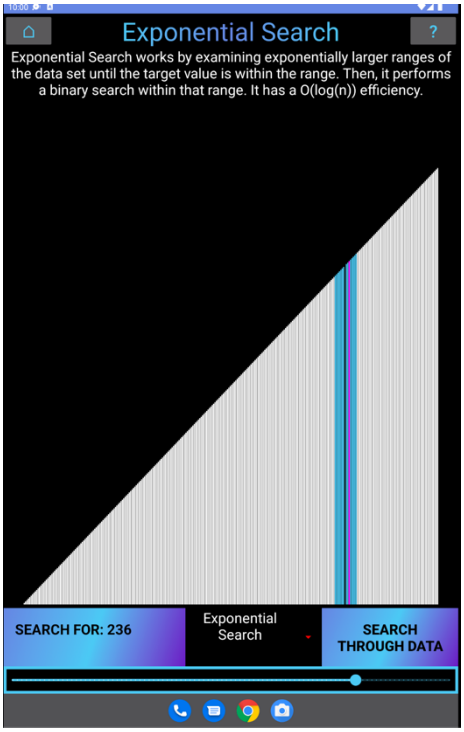


Figure 8

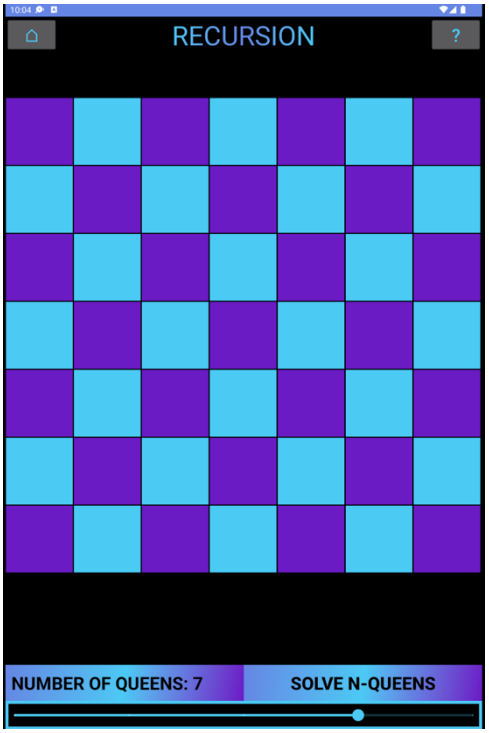


Figure 9

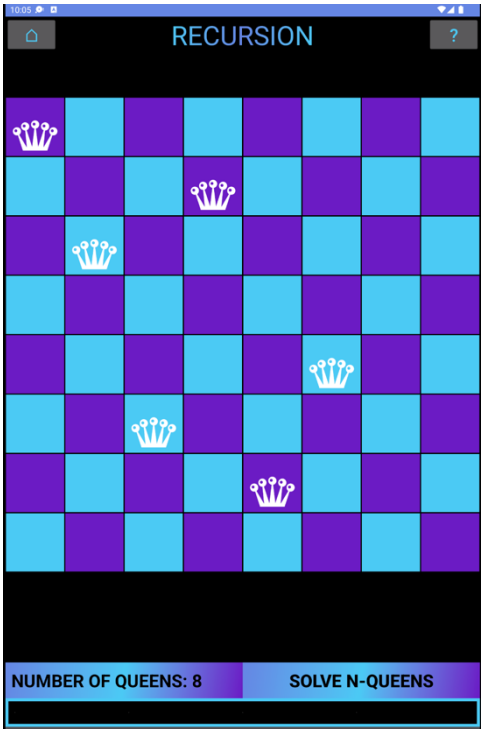


Figure 10

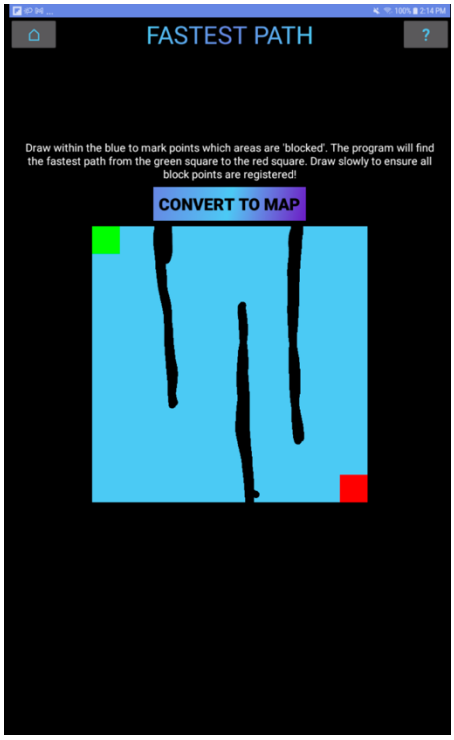


Figure 11

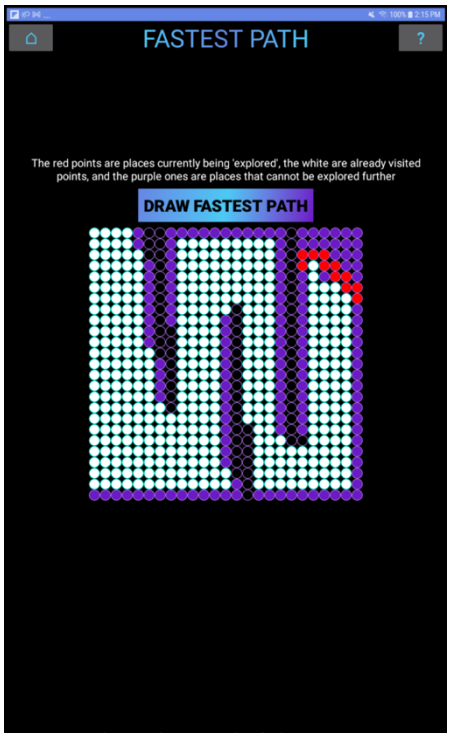


Figure 12