SORTING VISUALIZATION

Ha Noi University of Science and Technology

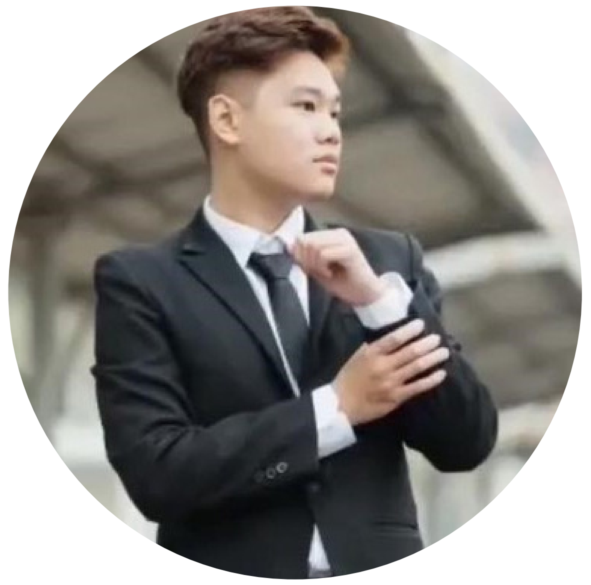
Object-oriented Programming Project

MEMBER LIST

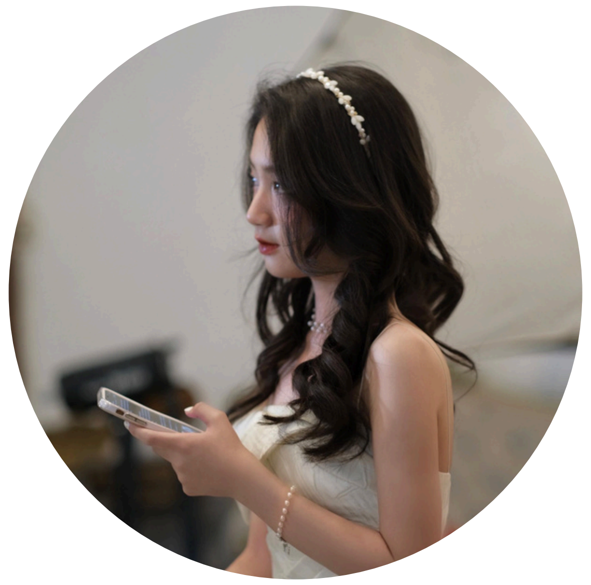
NGUYỄN MỸ DUYÊN

NGUYỄN ĐÌNH DƯƠNG

NGUYỄN LAN NHI



HÀ VIỆT KHÁNH



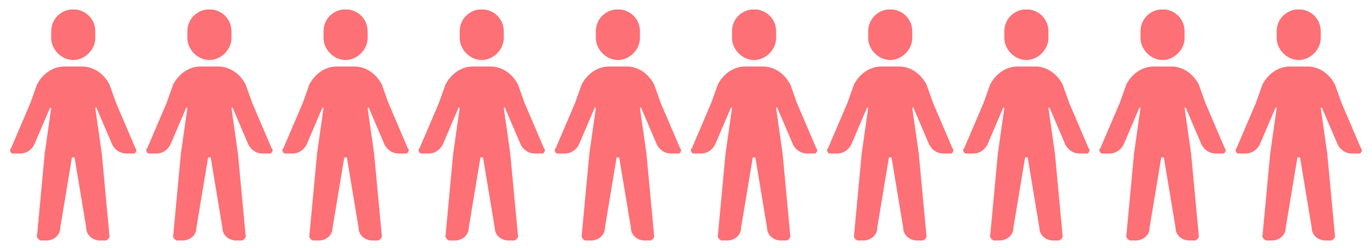
HỒ BẢO THƯ

NGUYỄN TR.MINH PHƯƠNG

PROJECT TIMELINE

1. Problems
2. Features
3. Class Diagrams
4. Techniques
5. Sorting Algorithms
6. Demo

PROBLEMS



ALL OF THE STUDENT OF IT MAJOR

Request for a detailed elucidation of fundamental sorting algorithms.

Visualize sorting algorithms on an array, making them become more intuitive

Virtual Assistant for use to asking questions related to sorting prob

Interactive Actions: Allowing users to engage with the algorithms

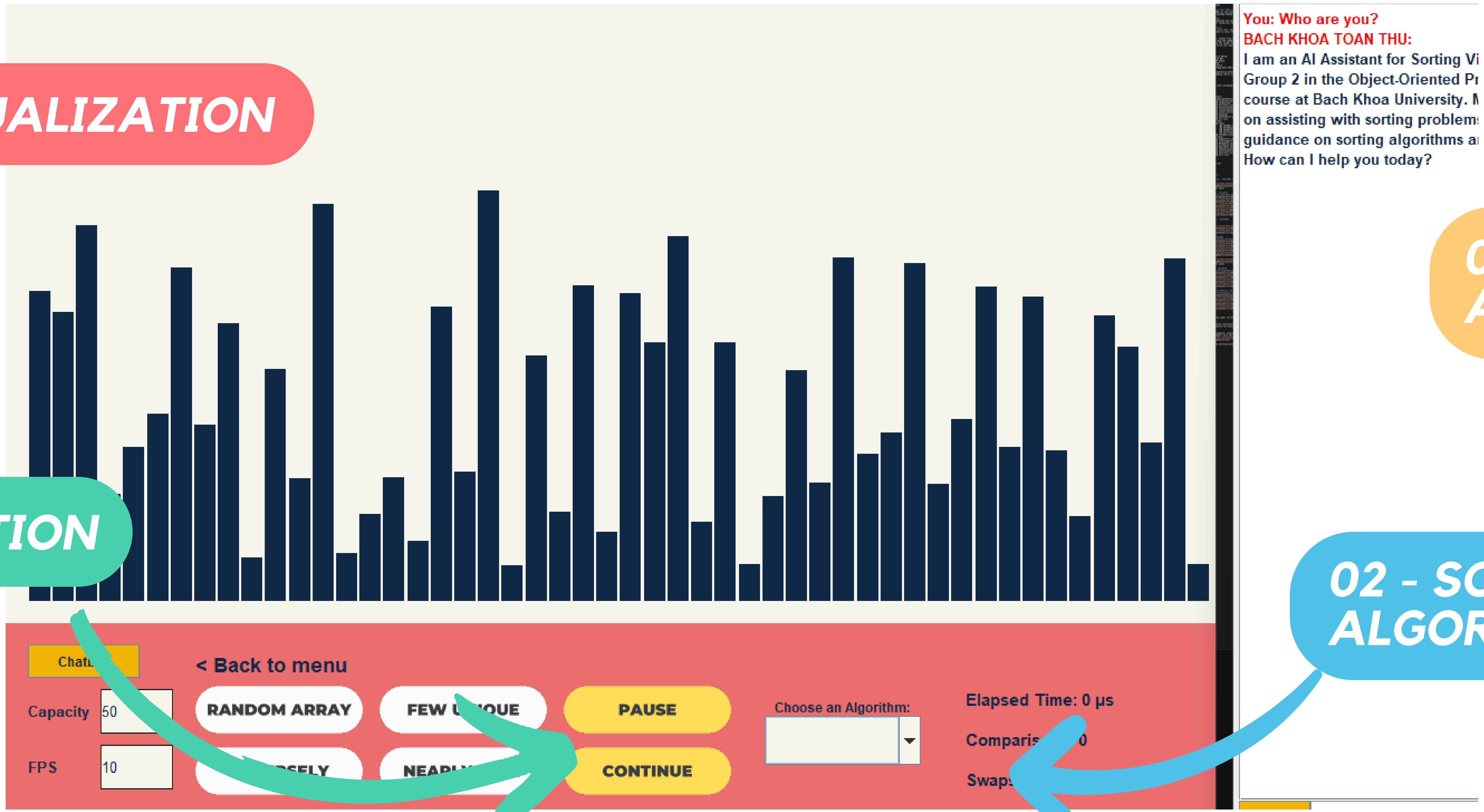
FEATURES AND TOOLS

01 - VISUALIZATION

02 - INTERACTION

04 -
AI ASSISTANT

02 - SORTING
ALGORITHMS



Capacity 50

FPS 10

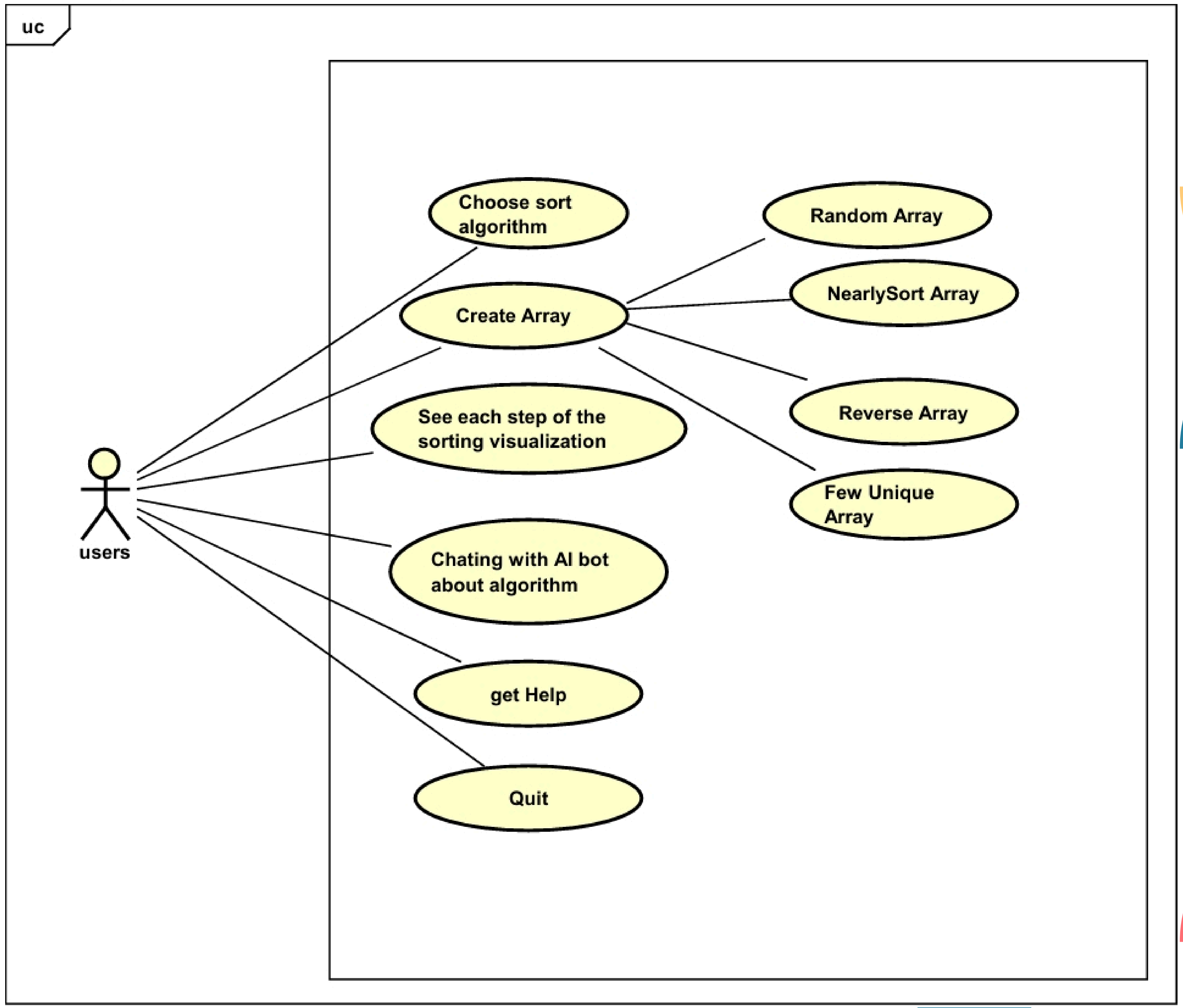
Elapsed Time: 0 µs

Comparis 0

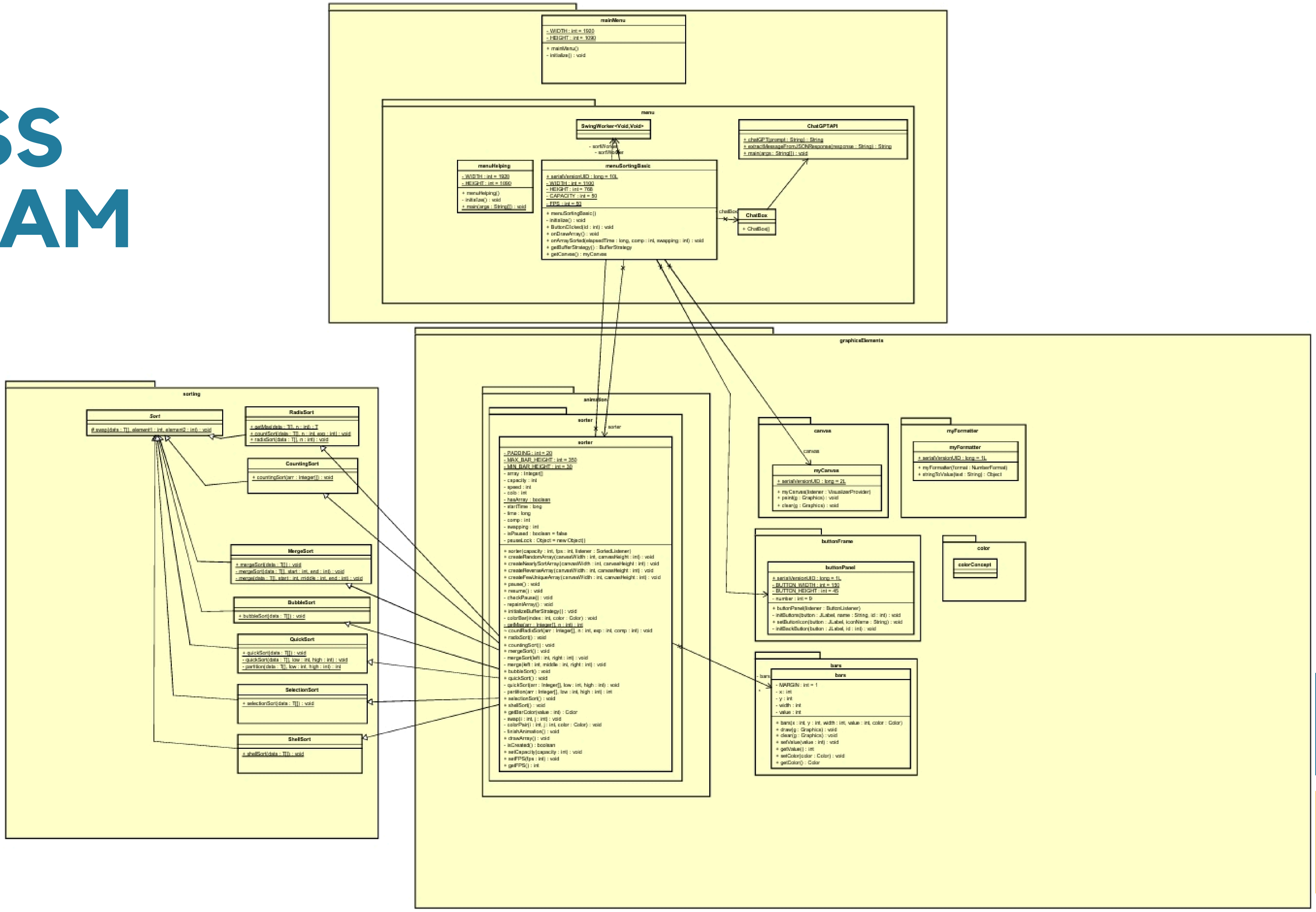
Swaps

You: Who are you?
BACH KHOA TOAN THU:
I am an AI Assistant for Sorting Vi
Group 2 in the Object-Oriented Pr
course at Bach Khoa University. M
on assisting with sorting problem
guidance on sorting algorithms a
How can I help you today?

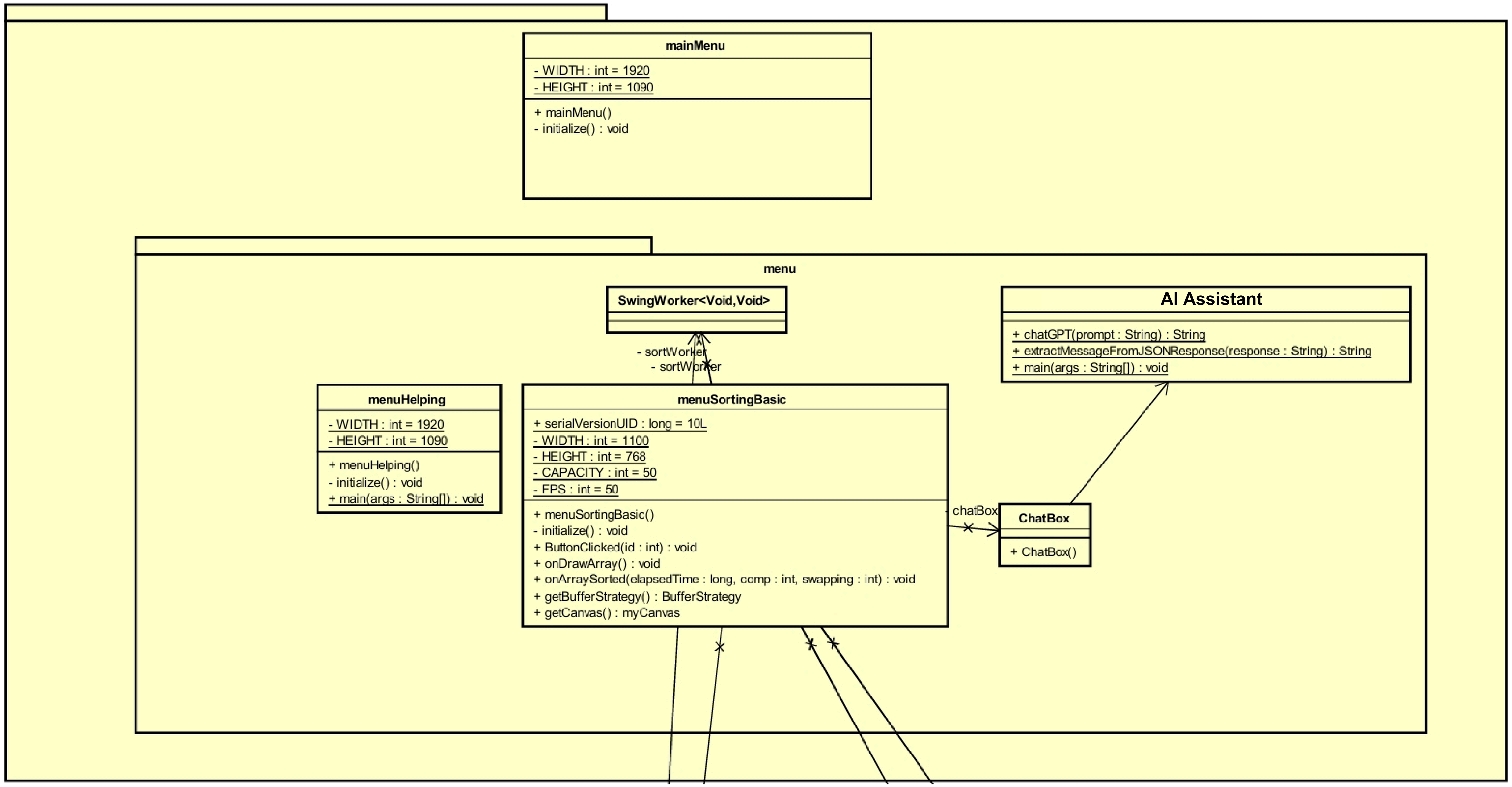
USE CASE DIAGRAM



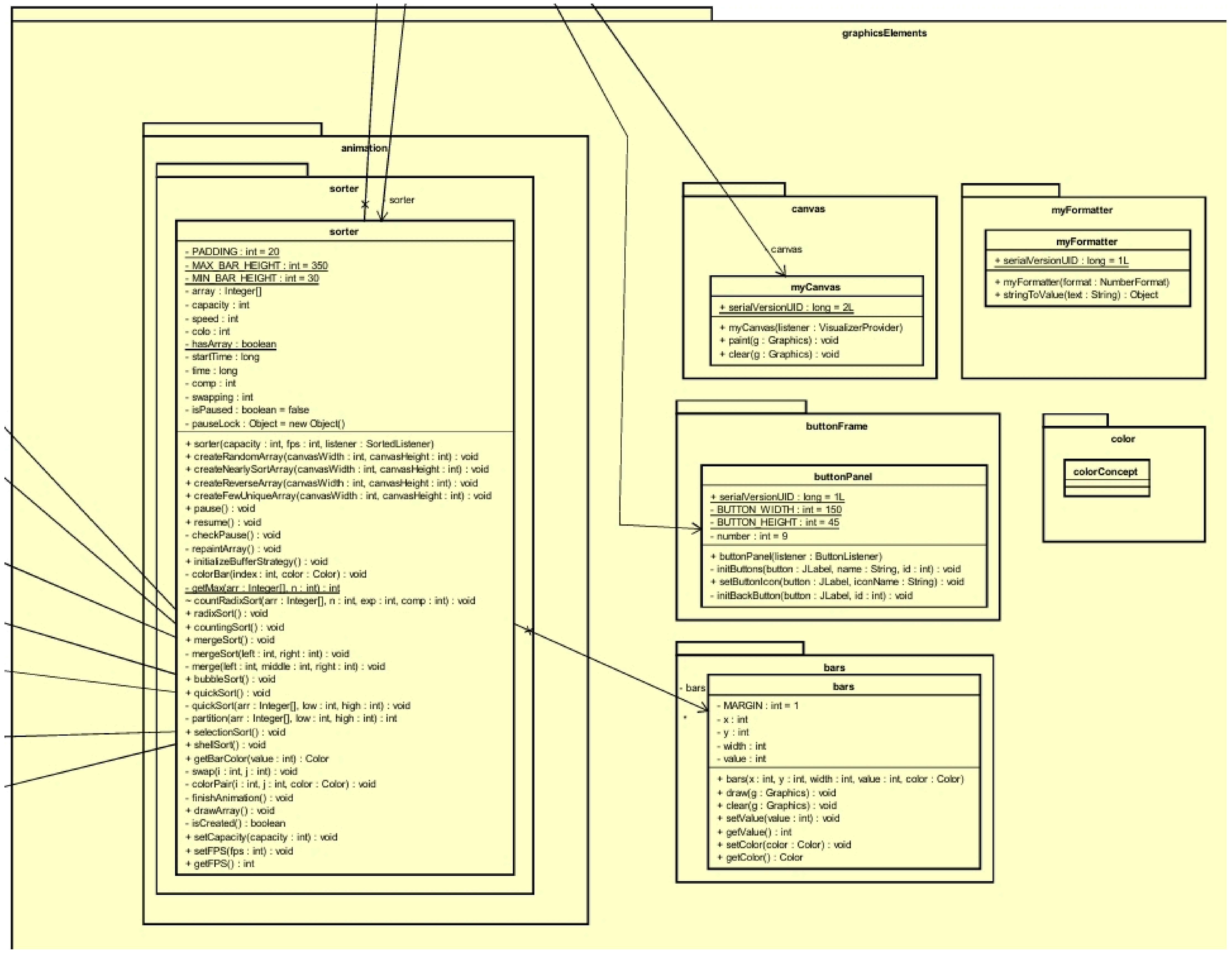
CLASS DIAGRAM


mainMenu
- WIDTH : int = 1920
- HEIGHT : int = 1090
+ mainMenu()
- initialize() : void
menu
SwingWorker<Void,Void>
- sortWorker
- sortWorker
ChatGPTAPI
+ chatGPT(prompt : String) : String
+ extractMessageFromJSONResponse(response : String) : String
+ main(args : String[]) : void
menuHelping
- WIDTH : int = 1920
- HEIGHT : int = 1090
+ menuHelping()
- initialize() : void
+ main(args : String[]) : void
menuSortingBasic
+ serialVersionUID : long = 10L
- WIDTH : int = 1920
- HEIGHT : int = 768
- CAPACITY : int = 50
- FPS : int = 50
+ menuSortingBasic()
- initialize() : void
+ ButtonClicked(id : int) : void
+ onDrawArray() : void
+ onArraySorted(elapsedTime : long, comp : int, swapping : int) : void
+ getBufferStrategy() : BufferStrategy
+ getCanvas() : myCanvas
- chatBox
ChatBox
+ ChatBox()
graphicsElements
sorting
Sort
# swap(data : T[], element1 : int, element2 : int) : void
RadixSort
+ getMax(data : T[], n : int) : T
+ countSort(data : T[], n : int, exp : int) : void
+ radixSort(data : T[], n : int) : void
CountingSort
+ countingSort(arr : Integer[]) : void
MergeSort
+ mergeSort(data : T[]) : void
- mergeSort(data : T[], start : int, end : int) : void
- merge(data : T[], start : int, middle : int, end : int) : void
BubbleSort
+ bubbleSort(data : T[]) : void
QuickSort
+ quickSort(data : T[]) : void
- quickSort(data : T[], low : int, high : int) : void
- partition(data : T[], low : int, high : int) : int
SelectionSort
+ selectionSort(data : T[]) : void
ShellSort
+ shellSort(data : T[]) : void
animation
sorter
- sorter
sorter
- PADDING : int = 20
- MAX_BAR_HEIGHT : int = 350
- MIN_BAR_HEIGHT : int = 30
- array : Integer[]
- capacity : int
- speed : int
- cols : int
- hasArray : boolean
- startTime : long
- time : long
- comp : int
- swapping : int
- isPaused : boolean = false
- pauseLock : Object = new Object()
+ sorter(capacity : int, fps : int, listener : SortedListener)
+ createRandomArray(canvasWidth : int, canvasHeight : int) : void
+ createNearlySortArray(canvasWidth : int, canvasHeight : int) : void
+ createReverseArray(canvasWidth : int, canvasHeight : int) : void
+ createFewUniqueArray(canvasWidth : int, canvasHeight : int) : void
+ pause() : void
+ resume() : void
- checkPause() : void
- repaintArray() : void
+ initializeBufferStrategy() : void
- colorBar(index : int, color : Color) : void
- getMax(arr : Integer[], n : int) : int
- countRadixSort(arr : Integer[], n : int, exp : int, comp : int) : void
+ radixSort() : void
+ countingSort() : void
+ mergeSort() : void
- mergeSort(left : int, right : int) : void
- merge(left : int, middle : int, right : int) : void
+ bubbleSort() : void
+ quickSort() : void
- quickSort(arr : Integer[], low : int, high : int) : void
- partition(arr : Integer[], low : int, high : int) : int
+ selectionSort() : void
+ shellSort() : void
+ getBarColor(value : int) : Color
- swap(i : int, j : int) : void
- colorPair(i : int, j : int, color : Color) : void
- finishAnimation() : void
+ drawArray() : void
- isCreated() : boolean
+ setCapacity(capacity : int) : void
+ setFPS(fps : int) : void
+ getFPS() : int
canvas
- canvas
myCanvas
+ serialVersionUID : long = 2L
+ myCanvas(listener : VisualizerProvider)
+ paint(g : Graphics) : void
+ clear(g : Graphics) : void
myFormatter
myFormatter
+ serialVersionUID : long = 1L
+ myFormatter(format : NumberFormat)
+ stringToValue(text : String) : Object
buttonFrame
buttonPanel
+ serialVersionUID : long = 1L
- BUTTON_WIDTH : int = 150
- BUTTON_HEIGHT : int = 45
- number : int = 9
+ buttonPanel(listener : ButtonListener)
- initButtons(button : JLabel, name : String, id : int) : void
+ setButtonIcon(button : JLabel, iconName : String) : void
- initBackButton(button : JLabel, id : int) : void
color
colorConcept
bars
- bars
*
bars
- MARGIN : int = 1
- x : int
- y : int
- width : int
- value : int
+ bars(x : int, y : int, width : int, value : int, color : Color)
+ draw(g : Graphics) : void
+ clear(g : Graphics) : void
+ setValue(value : int) : void
+ getValue() : int
+ setColor(color : Color) : void
+ getColor() : Color

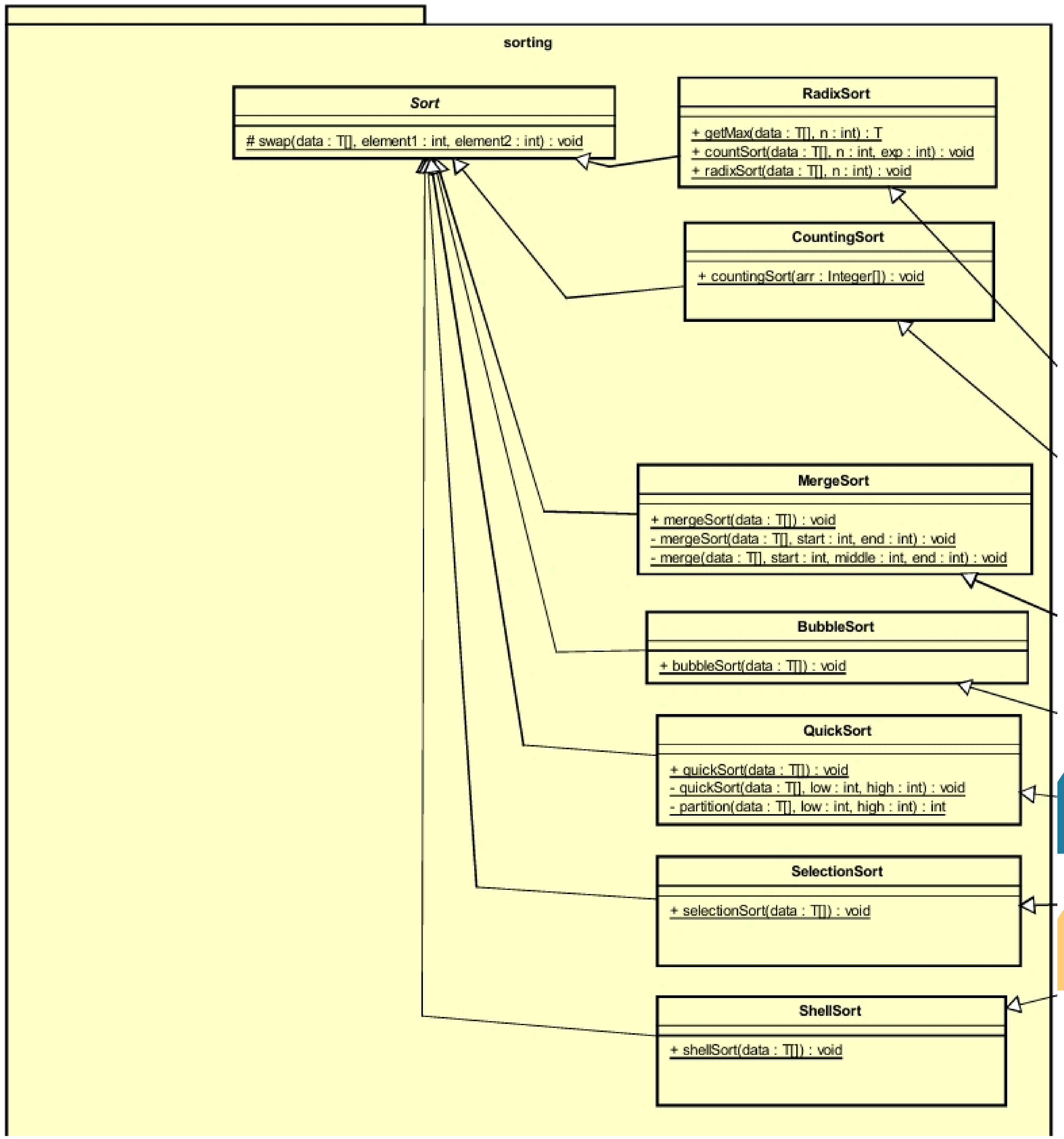
CLASS DIAGRAM



CLASS DIAGRAM


graphicsElements
animation
sorter
sorter
sorter
- PADDING : int = 20
- MAX_BAR_HEIGHT : int = 350
- MIN_BAR_HEIGHT : int = 30
- array : Integer[]
- capacity : int
- speed : int
- colo : int
- hasArray : boolean
- startTime : long
- time : long
- comp : int
- swapping : int
- isPaused : boolean = false
- pauseLock : Object = new Object()
+ sorter(capacity : int, fps : int, listener : SortedListener)
+ createRandomArray(canvasWidth : int, canvasHeight : int) : void
+ createNearlySortArray(canvasWidth : int, canvasHeight : int) : void
+ createReverseArray(canvasWidth : int, canvasHeight : int) : void
+ createFewUniqueArray(canvasWidth : int, canvasHeight : int) : void
+ pause() : void
+ resume() : void
- checkPause() : void
- repaintArray() : void
+ initializeBufferStrategy() : void
- colorBar(index : int, color : Color) : void
- getMax(arr : Integer[], n : int) : int
~ countRadixSort(arr : Integer[], n : int, exp : int, comp : int) : void
+ radixSort() : void
+ countingSort() : void
+ mergeSort() : void
- mergeSort(left : int, right : int) : void
- merge(left : int, middle : int, right : int) : void
+ bubbleSort() : void
+ quickSort() : void
- quickSort(arr : Integer[], low : int, high : int) : void
- partition(arr : Integer[], low : int, high : int) : int
+ selectionSort() : void
+ shellSort() : void
+ getBarColor(value : int) : Color
- swap(i : int, j : int) : void
- colorPair(i : int, j : int, color : Color) : void
- finishAnimation() : void
+ drawArray() : void
- isCreated() : boolean
+ setCapacity(capacity : int) : void
+ setFPS(fps : int) : void
+ getFPS() : int
canvas
canvas
myCanvas
+ serialVersionUID : long = 2L
+ myCanvas(listener : VisualizerProvider)
+ paint(g : Graphics) : void
+ clear(g : Graphics) : void
myFormatter
myFormatter
+ serialVersionUID : long = 1L
+ myFormatter(format : NumberFormat)
+ stringToValue(text : String) : Object
buttonFrame
buttonPanel
+ serialVersionUID : long = 1L
- BUTTON_WIDTH : int = 150
- BUTTON_HEIGHT : int = 45
- number : int = 9
+ buttonPanel(listener : ButtonListener)
- initButtons(button : JLabel, name : String, id : int) : void
+ setButtonIcon(button : JLabel, iconName : String) : void
- initBackButton(button : JLabel, id : int) : void
color
colorConcept
bars
- bars
*
bars
- MARGIN : int = 1
- x : int
- y : int
- width : int
- value : int
+ bars(x : int, y : int, width : int, value : int, color : Color)
+ draw(g : Graphics) : void
+ clear(g : Graphics) : void
+ setValue(value : int) : void
+ getValue() : int
+ setColor(color : Color) : void
+ getColor() : Color

CLASS DIAGRAM


sorting
Sort
# swap(data : T[], element1 : int, element2 : int) : void
RadixSort
+ getMax(data : T[], n : int) : T
+ countSort(data : T[], n : int, exp : int) : void
+ radixSort(data : T[], n : int) : void
CountingSort
+ countingSort(arr : Integer[]) : void
MergeSort
+ mergeSort(data : T[]) : void
- mergeSort(data : T[], start : int, end : int) : void
- merge(data : T[], start : int, middle : int, end : int) : void
BubbleSort
+ bubbleSort(data : T[]) : void
QuickSort
+ quickSort(data : T[]) : void
- quickSort(data : T[], low : int, high : int) : void
- partition(data : T[], low : int, high : int) : int
SelectionSort
+ selectionSort(data : T[]) : void
ShellSort
+ shellSort(data : T[]) : void

OBJECT-ORIENTED TECHNIQUES

OOP TECHNIQUES

INHERITANCE

- MergeSort, CountingSort, RadixSort, BubbleSort, SelectionSort, ShellSort, QuickSort inherit from Sort
- myFormatter inherit from javax.swing.text.NumberFormatter
- buttonPanel inherit from javax.swing.JPanel
- myCanvas inherit from java.awt.Canvas
- mainMenu, menuSortingBasic, ChatBox inherit from javax.swing.JFrame

OVERIDING

- myCanvas overrides paint() method of java.awt.Canvas
- myFormatter overrides stringToValue() method of javax.swing.text.InternationalFormatter

OOP TECHNIQUES

AGGREGATION

- sorter has a bars, sort algorithms, myCanvas,...
- sorter use createRandomArray, createNearlySortArray, createReverseArray, createFewUniqueArray, Sort button
- mainMenu has menuSortingBasic, menuHelping
- menuSortingBasic has ChatBox

INTERFACE

- buttonPanel interacts with menuSortingBasic via the ButtonListener interface
- myCanvas interacts with menuSortingBasic via the VisualizerProvider interface
- sorter interacts with menuSortingBasic via the SortedListener interface

SORTING ALGORITHMS

BUBBLE SORT

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.

BEST CASE

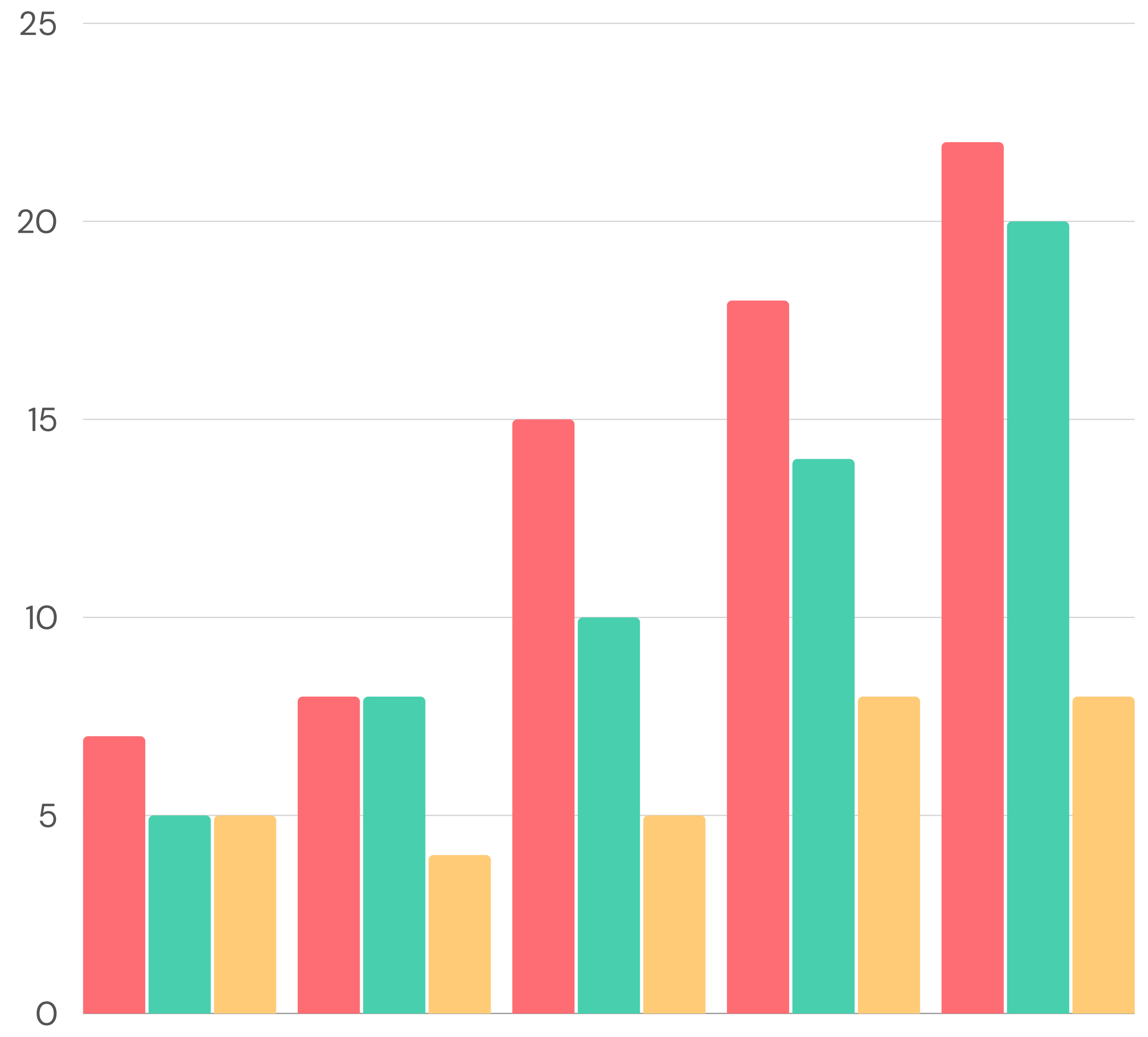
O(n) (when the list is already sorted)

AVERAGE CASE

O(n^2)

WORST CASE

The worst-case scenario for Bubble Sort occurs when the list is sorted in reverse order. In this case, the algorithm has to perform the maximum number of comparisons and swaps. Each element needs to be compared with every other element, resulting in a time complexity of O(n2)



COUNTING SORT

Counting Sort is a non-comparison-based sorting algorithm that works well when there is limited range of input values. It is particularly efficient when the range of input values is small compared to the number of elements to be sorted. The basic idea behind Counting Sort is to count the frequency of each distinct element in the input array and use that information to place the elements in their correct sorted positions.

BEST CASE

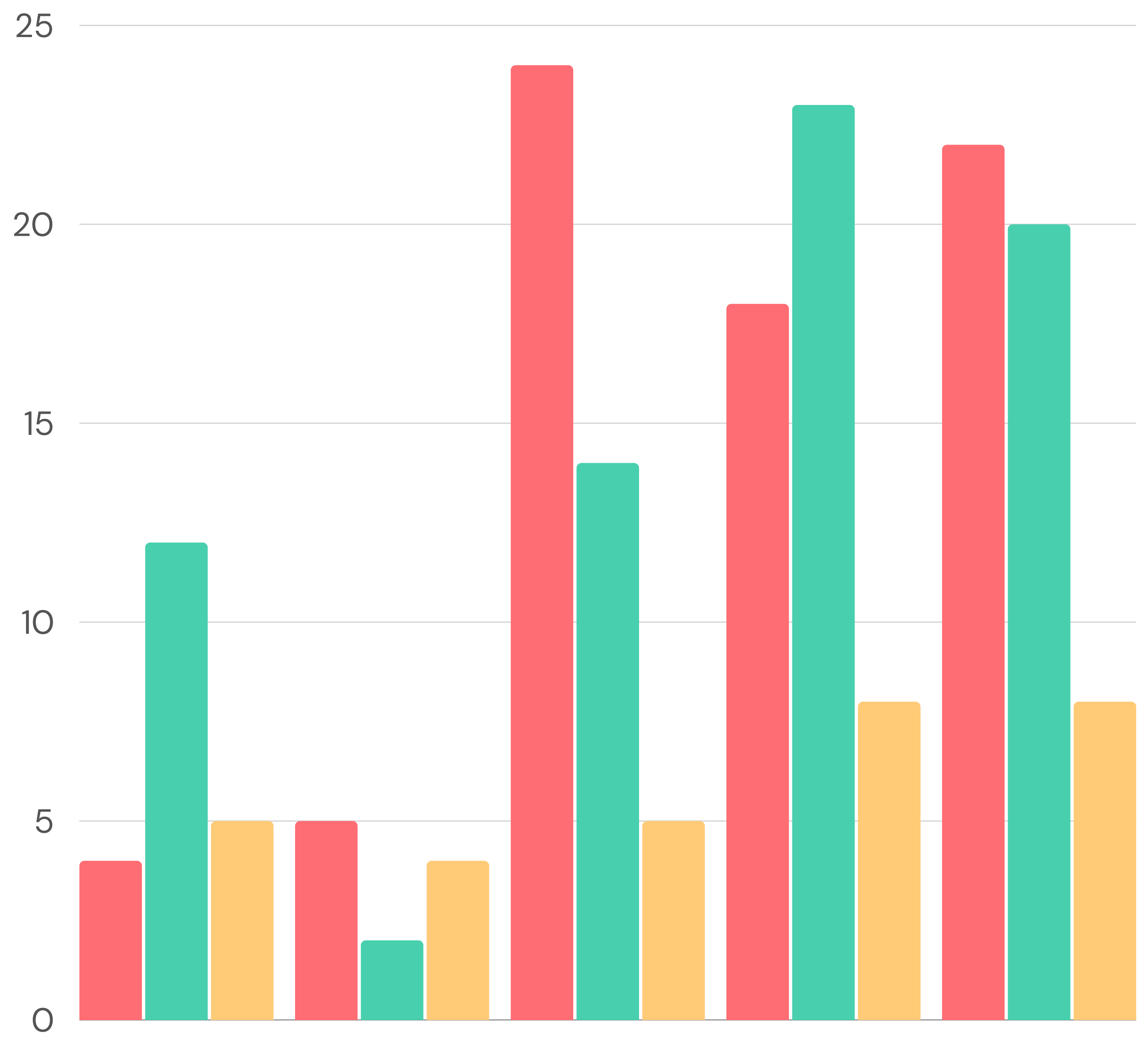
Counting Sort is efficient for sorting integers or objects that can be converted to integers, especially when the range of input values k is not significantly larger than the number of elements nnn.

AVERAGE CASE

O(n+k)
Here, n is the number of elements, and k is the range of the input values.

WORST CASE

The worst-case scenario for Counting Sort occurs when the range of input values k is significantly larger than the number of elements n. In this case, the time and space complexity can become inefficient due to the large size of the auxiliary count array.



RADIX SORT

Radix Sort is a linear sorting algorithm that sorts elements by processing them digit by digit. It is an efficient sorting algorithm for integers or strings with fixed-size keys.

BEST CASE

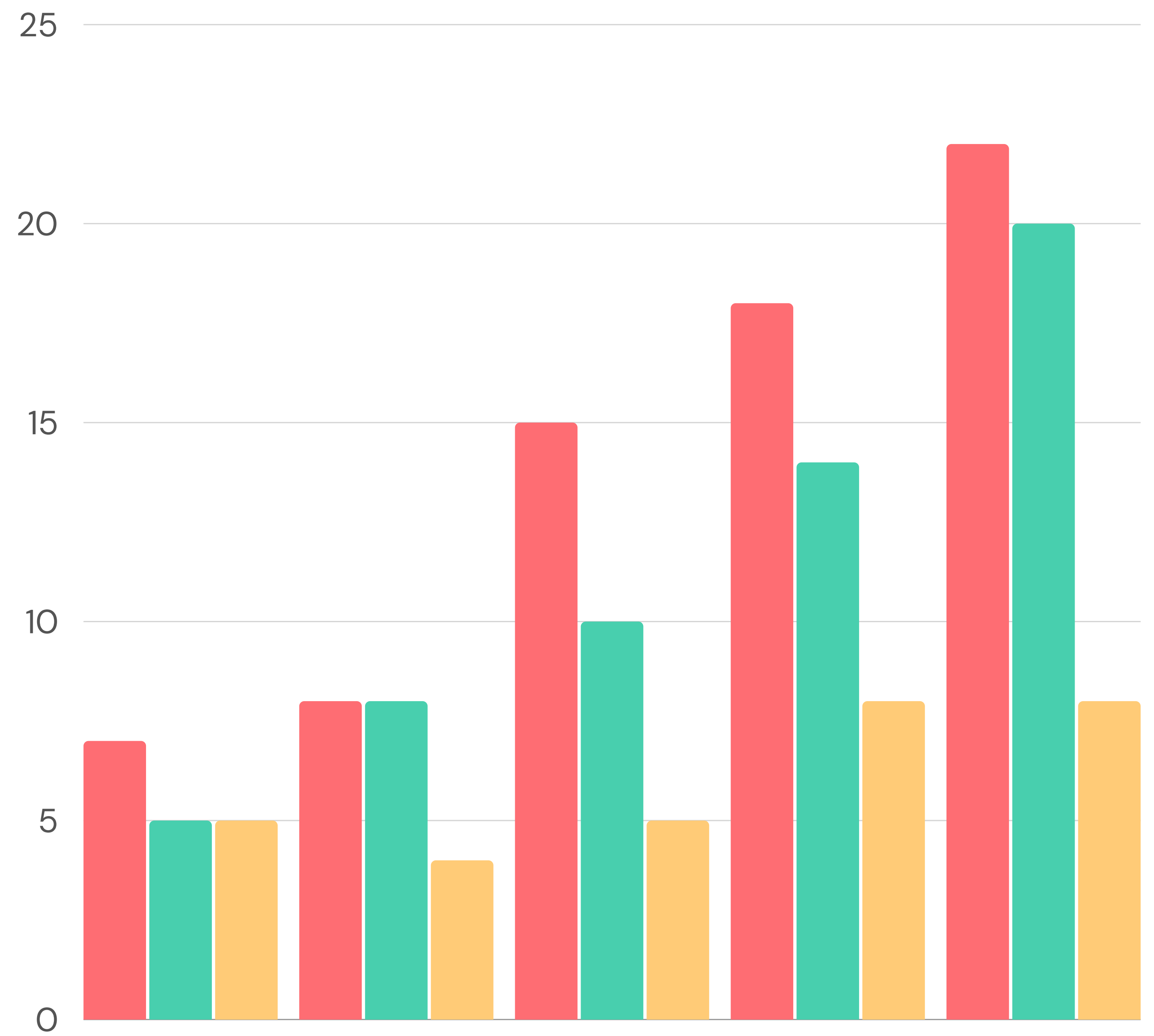
The best case for Radix Sort occurs when the number of digits d is small and the range of digits k is constant. Since Radix Sort processes each digit independently, it performs consistently regardless of the initial order of elements.

AVERAGE CASE

O(nk)

WORST CASE

Large Number of Digits: If the numbers have a large number of digits, the factor d in the complexity O(d×(n+k)) becomes significant.
Large Digit Range: If the digit range k were large (which is rare in practical applications), it could also affect the efficiency.



QUICK SORT

QuickSort is a sorting algorithm based on the Divide and Conquer algorithm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

BEST CASE

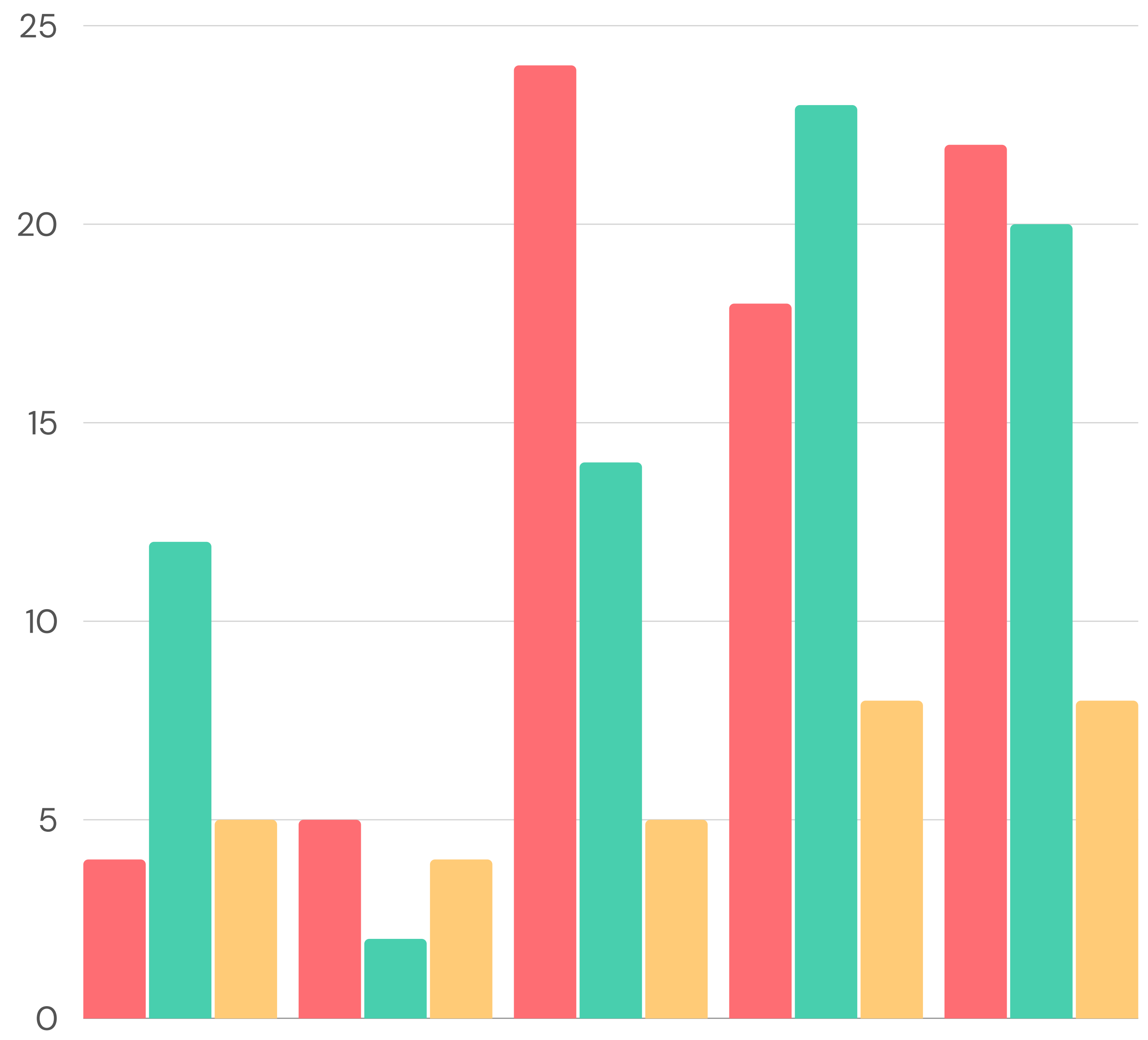
O(nlogn).The best-case scenario for Quick Sort occurs when the pivot chosen at each step perfectly partitions the array into two equal halves. This ensures that the depth of the recursive tree is minimized, resulting in the most balanced possible divide-and-conquer process.

AVERAGE CASE

O(nlogn)

WORST CASE

O(n2) (occurs when the smallest or largest element is always chosen as the pivot)



SELECTION SORT

Selection sort is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list.

BEST CASE

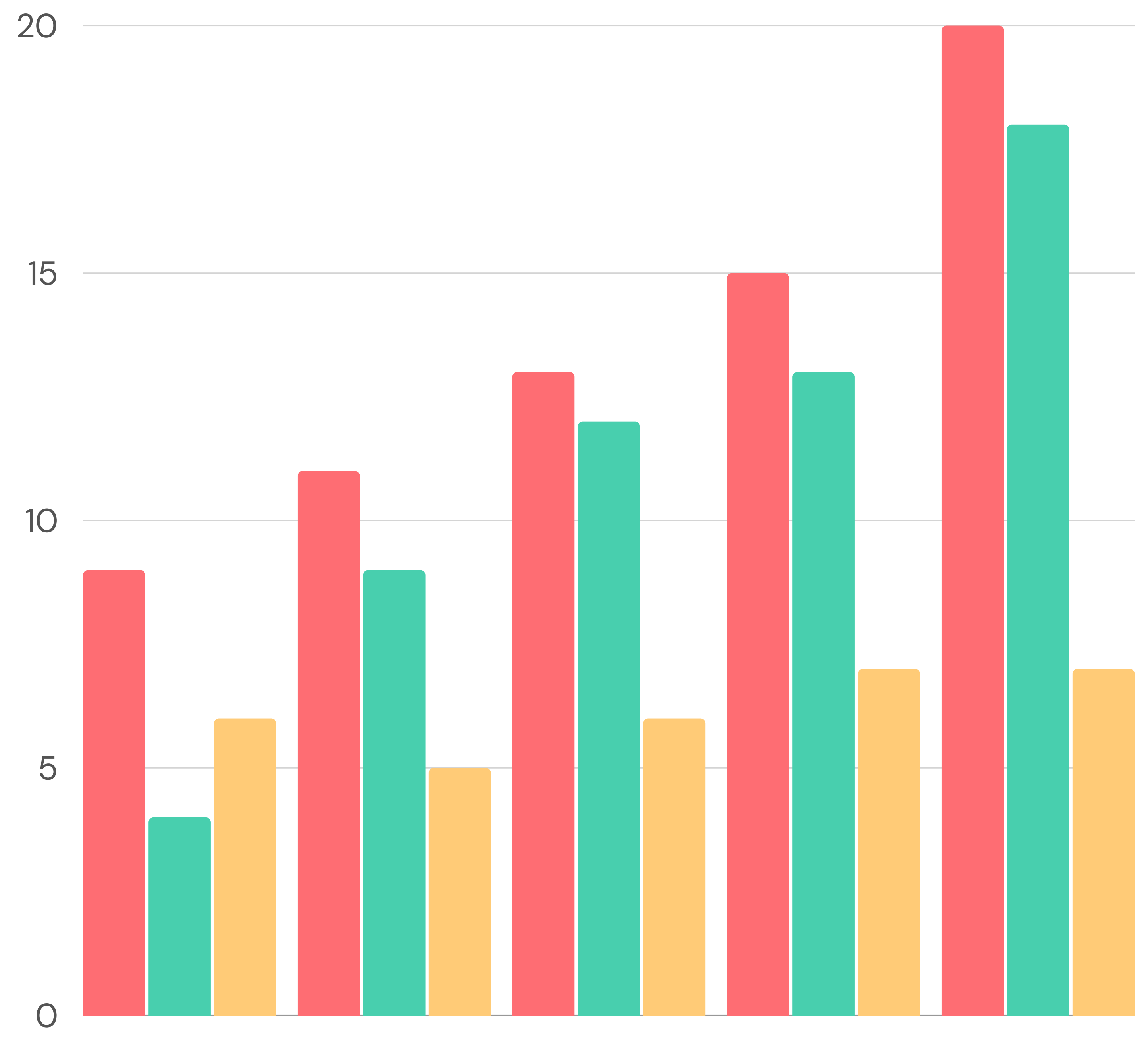
Already Sorted Sequences: Selection Sort does not take advantage of the fact that the sequence is already sorted, so its performance remains O(n^2).

AVERAGE CASE

Random Sequences: The performance of Selection Sort on a random sequence will also be O(n^2), with no particular advantage or disadvantage.

WORST CASE

Reversed Sequences: Similar to sorted sequences, Selection Sort will still perform O(n^2) operations since it needs to compare each element.



MERGE SORT

Merge sort is a sorting algorithm that follows the divide-and-conquer approach. It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array.

BEST CASE

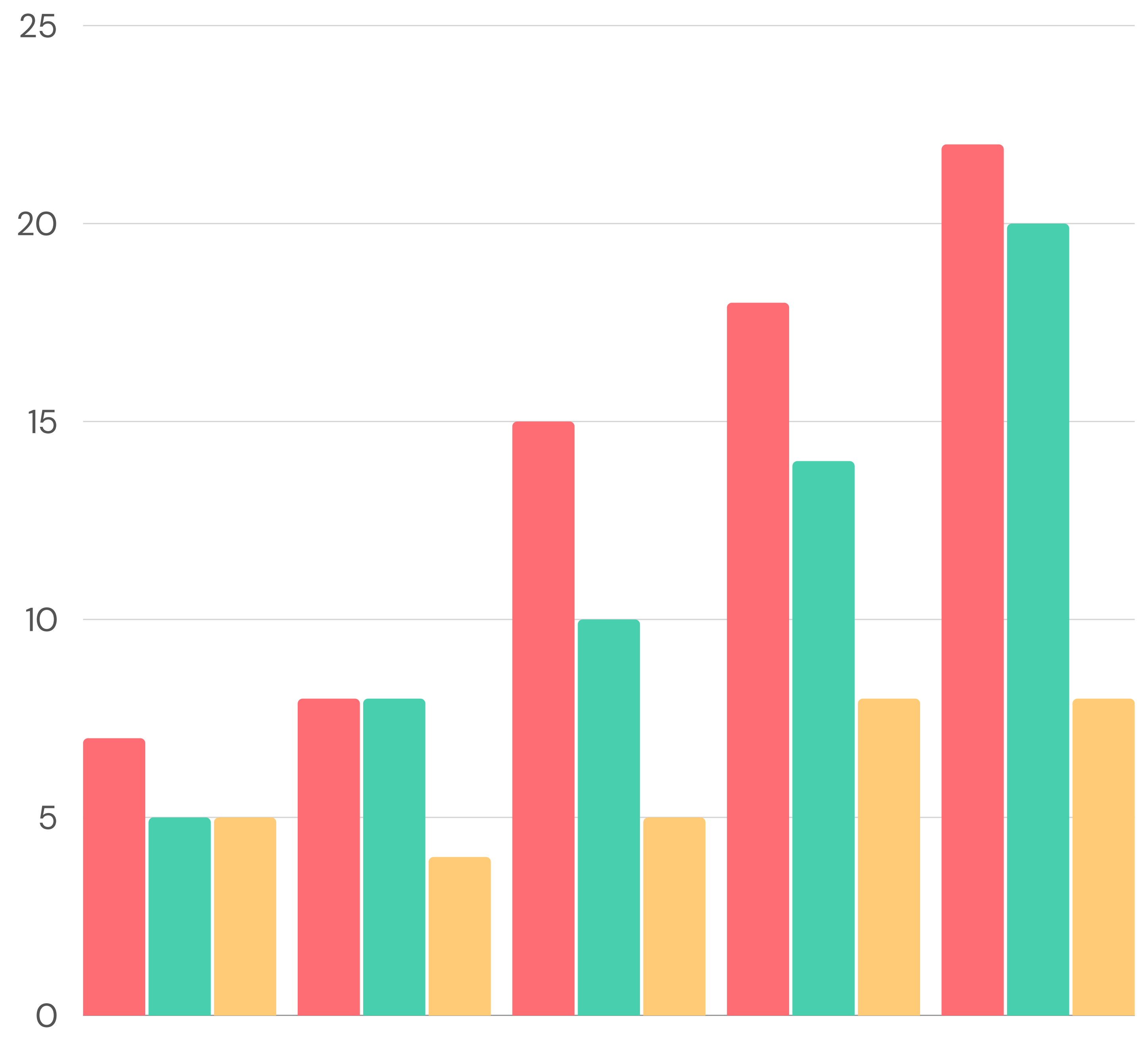
Already Sorted Sequences: Merge Sort does not take advantage of the fact that the sequence is already sorted, so its performance remains O(n*log n)

AVERAGE CASE

Random Sequences: The performance of Selection Sort on a random sequence will also be O(n*log n), with no particular advantage or disadvantage.

WORST CASE

Reversed Sequences: Similar to sorted sequences, Selection Sort will still perform O(n*log n) operations since it needs to compare each element.



SHELL SORT

Shell sort is an in-place comparison sort that generalizes insertion sort to allow the exchange of items that are far apart, using a sequence of gaps to gradually reduce the distance between compared elements.

BEST CASE

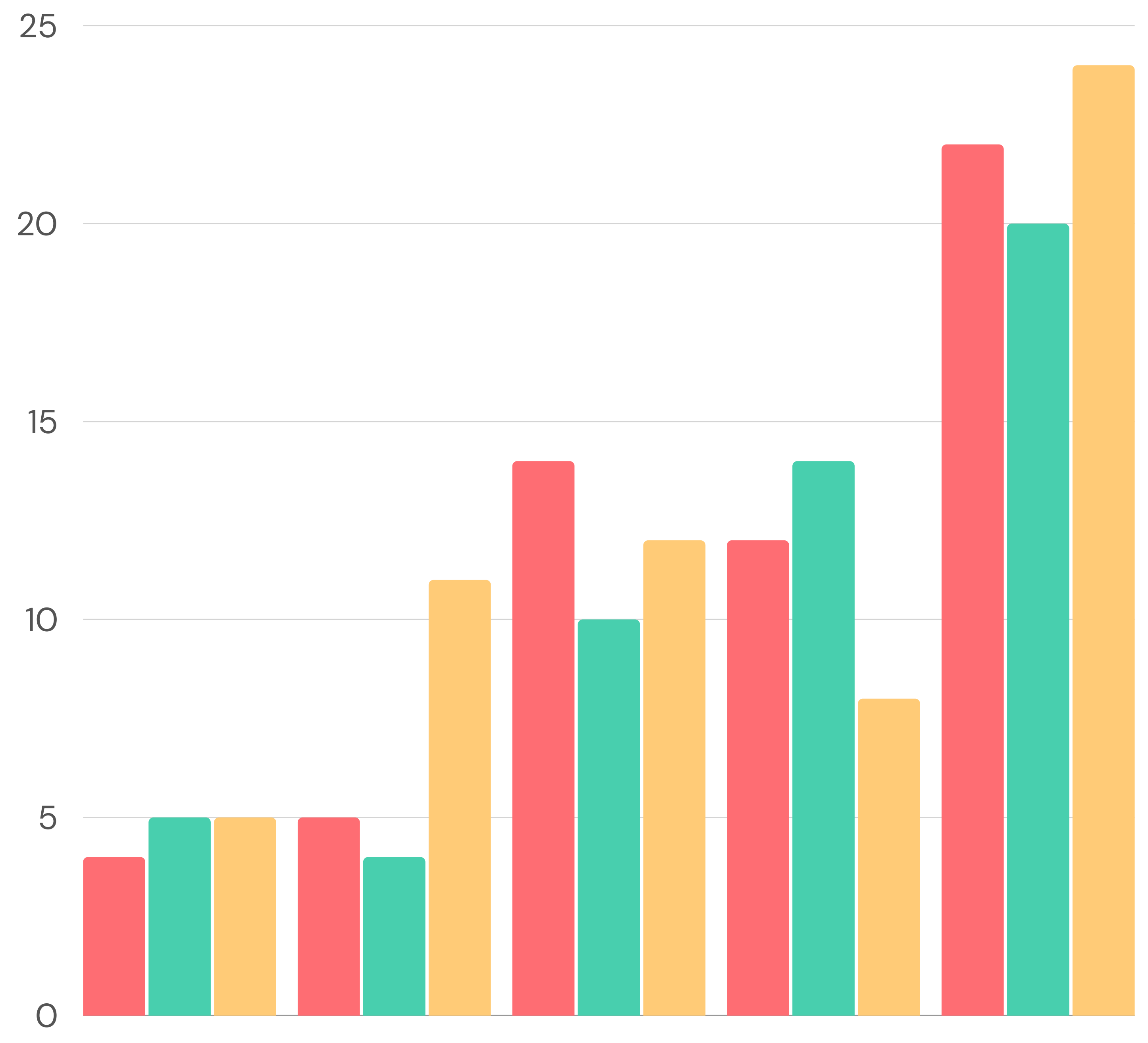
When the given array list is already sorted the total count of comparisons of each interval is equal to the size of the given array.
So best case complexity is O(n*log(n))

AVERAGE CASE

The Average Case Complexity: O(n*log n)~O(n^1.25)

WORST CASE

The worst type of sequence for Shell sort is where the gaps are simply halved each time (e.g., n/2, n/4, n/8, ..., 1). This sequence can result in a worst-case time complexity of (n^2), which is significantly less efficient than more advanced gap sequences.



DEMO

THANK YOU