Criteria A

Scenario (client and problem)

My client is Mr. Ismith, an IB Computer Science teacher at Overseas Family School (OFS) who teaches both HL and SL Computer Science. In his teaching, he regularly introduces students to fundamental algorithms such as sorting and searching, and he often needs to demonstrate how these algorithms behave on different types of datasets. This includes showing efficiency, time complexity, behaviour on sorted vs. random data, and the practical differences between algorithms like Quick Sort and Bubble Sort, or Linear Search and Binary Search.

During our initial consultation (see Interview Transcript – Appendix A), the client explained that his current method of comparing algorithms is extremely inefficient. At present, he:

- Runs each algorithm separately in different Java files, Inserts manual timing code using System.nanoTime();
- only discusses the theoretical Big-O complexity without any systematic empirical tests.
- Copies results into spreadsheets to make comparisons;
- Must edit and recompile code whenever trying a new dataset;
- Cannot easily load external CSV data;
- Must re-run every demonstration separately for each class.

This approach is time-consuming, inconsistent, error-prone and not reusable across different teaching sessions. He also cannot easily show students how algorithms behave on datasets with different characteristics (sorted, nearly sorted, reverse sorted, random). Nor can he quickly demonstrate how complexity such as $O(n^2)$ vs $O(n \log n)$ manifests in practice. This makes it difficult for the client to provide accurate demonstrations of algorithm performance, especially when comparing multiple sorting algorithms (such as Bubble, Merge, Quick and Heap Sort) or search algorithms (Linear vs Binary Search) on realistic data from CSV files (Client Interview, Q2–Q4).

The core problem identified by the client is:
He lacks a unified, reusable benchmarking system that can load datasets, allow users to choose multiple algorithms, run them fairly on identical inputs, measure execution time, count comparisons/iterations, classify the dataset, and generate clear results for teaching purposes.

The client's current "legacy system": a set of unrelated Java files, print statements, and spreadsheet recordings of times, confirming the system is non-automated, manual, and unsuitable for repeated use.

To address this, I will design and develop a Java-based Algorithm Benchmarking System (ABS) that automates the tasks the client currently performs manually.

Current product

Legacy system: a set of unrelated Java files, print statements, and spreadsheet recordings of times. No actual system present.

Proposed product and justification

I propose to develop a console-based Java application called the Algorithm Benchmarking System (ABS). The ABS will allow the client or his students to:

- Choose a dataset (preset or imported from CSV);
- Choose multiple algorithms (sorting and searching) to compare;
- Automatically measure performance (time, loop counts, variability);
- Display and export the results in a clear, ranked form.
- Optionally export the data

Why this product solves the problem

1) Centralised, repeatable benchmarking
    a) Instead of writing separate timing code for each experiment, the client will use one tool. A core component, BenchmarkEngine, will handle timing, repeated runs and statistical calculations. This makes tests consistent across datasets, algorithms, classes and years.
2) Supports real teaching use cases
    a) The system directly supports the IB CS syllabus focus on algorithmic thinking, complexity and data structures. It allows the client to run live demonstrations in class, and students can repeat the experiments themselves using the same interface, reinforcing both theory and practice.
3) Handles different dataset types
    a) The ABS will ship with several preset datasets (random, nearly sorted, reverse sorted) and will load integer data from CSV. This allows the client to use real-world data or custom examples in addition to the built-in sets.
4) Detailed, objective measurements
    a) The benchmarking engine will record precise timing using System.nanoTime(), number of runs, number of loop iterations/comparisons and standard deviation. This gives deeper insight than the client's current "single time reading", and it helps students understand the variability and fairness of experiments
5) Big-O plus empirical comparison
    a) Each algorithm object will store its theoretical complexity (e.g. O(n), O(n log n), $O(n^2)$). After benchmarking, the system will generate a short comparison summary linking Big-O to the measured results. This addresses a key conceptual gap identified by the client: how to connect theory with observed behaviour.
6) Export and reuse of data
    a) The ABS will export results to CSV so that the client can quickly plot graphs or do further analysis in tools such as Excel or Google Sheets. It will also provide an option to export the sorted dataset produced by the fastest sorting algorithm.

Choice of tools and environment

- Programming language: Java 17 — already used in IB CS, supports precise timing, strong typing and object-oriented design. Matches the client's curriculum and teaching environment (Client Interview, Q1). Supports strong OOP principles needed for algorithm abstraction. Portable across all school devices
- IDE: IntelliJ IDEA— supports refactoring, debugging and JUnit testing. It is also free of cost for educational purposes.
- Diagramming tools: draw.io / diagrams.net for UML class diagrams, system flowcharts and decomposition diagrams, in line with the school's solution overview recommendations.
- Target platform: Standard OFS school laptops running Windows or macOS with Java installed, as confirmed by the client.

The ABS is clearly better than the existing solution, because it is faster, more reliable, easier to reuse, and purpose-built for teaching algorithm benchmarking. It also allows me to demonstrate original algorithmic thinking and design, especially in the BenchmarkEngine and input-validation modules, which are written from scratch.

The client's hardware and software where the proposed product will be implemented/run:

Processor	M4 Chip with 10 core CPU 16-core Neural Engine
Video	M4 Chip with 10 Core GPU
RAM	16GB Unified Memory
Storage	512GB SSD Storage
Display	Liquid Retina XDR display
Ports	Three Thunderbolt 4 Ports, HDMI Port, SDXC Card Slot, Headphone Jack
Keyboard	Backlit Magic Keyboard with Touch ID - US English

("MacBook Specs | Technology Support Services")

Success criteria

These criteria are made directly with the interview I had with my client, and I made sure to cover everything. This will be used later in Criterion E to evaluate the final product.

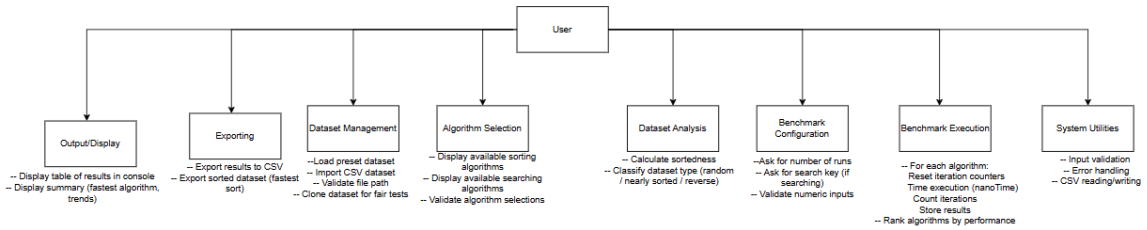
1. Preset datasets

1.1. The product must provide at least three built-in integer datasets: random, nearly sorted, and reverse sorted. Each must contain at least 100 elements and display its size and type to the user.

2. CSV import with validation
   2.1. The product must allow the user to import a dataset from a CSV file of integers. Non-integer values or empty files must be rejected with a clear error message, and the user must be prompted to re-enter a valid file path.
3. Time measurement
   3.1. For each algorithm, the system must record the execution time, for every run.
4. Algorithm selection
   4.1. The product must allow the user to select two or more algorithms from a menu, including at least: Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, Heap Sort, Linear Search and Binary Search. I might also include Interpolation search, jump search, shell sort and counting sort
5. Loop / iteration counting
   5.1. For algorithms that support it (e.g. Bubble Sort, Quick Sort), the system must track the number of key loop iterations or comparisons and report a total per run.
6. Dataset classification output
   6.1. For the chosen dataset, the system must display a classification such as “random”, “nearly sorted” or “reverse sorted” based on the order of elements (using a custom dataset-analysis algorithm).
7. Big-O comparison summary
   7.1. For each algorithm, the system must show its theoretical time complexity (e.g. $O(n)$, $O(n \log n)$, $O(n^2)$) and provide a short textual summary comparing theoretical and empirical performance, such as “Although Algorithm X is $O(n^2)$, it outperformed Algorithm Y on this dataset”.
8. Ranked results view
   8.1. After benchmarking, the product must display a ranked table listing algorithm name, time taken, loop count and number of runs, sorted from fastest to slowest, theoretical time complexity.
9. CSV export of results
   9.1. The user must be able to export all benchmark results to a CSV file, suitable for later graphing.
10. Export of sorted dataset (for sorting algorithms)
    10.1. For sorting benchmarks, the user must have the option to save the sorted dataset produced by the fastest algorithm to a CSV file.
11. Robust input validation
    11.1. All user inputs (menu choices, file paths, number of runs) must be validated. Invalid input must not cause the program to crash; instead, a clear error message must be shown and the user prompted again.
12. Console usability
    12.1. Clear menus, prompts, and labels so that a new user (e.g. another CS teacher) can run a basic benchmark without external instructions.
13. The system must be able to compute tasks until the user desires to quit/close out of the program.

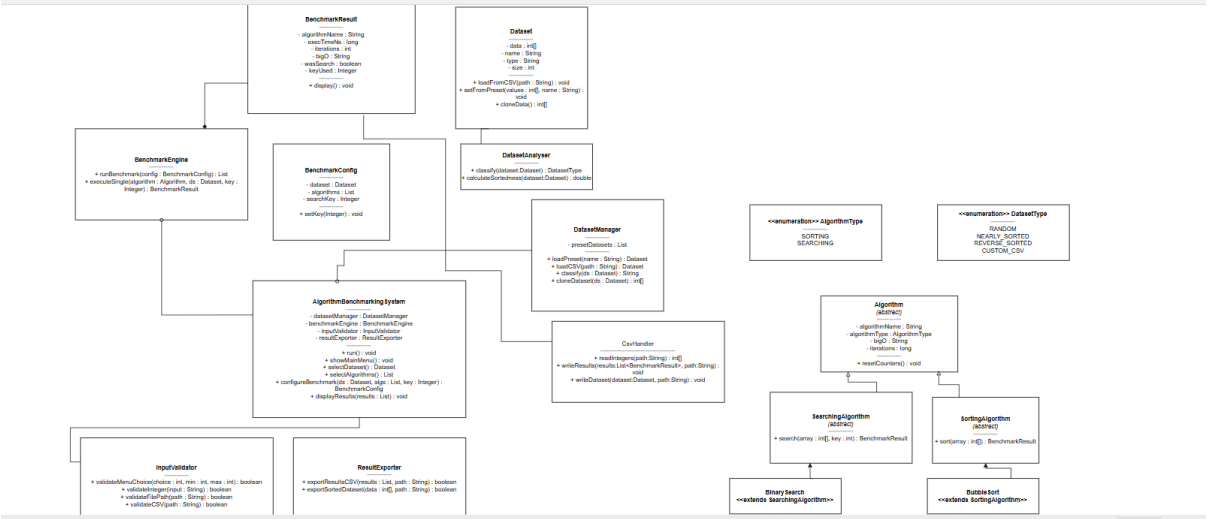
Criteria B

Decomposition diagram

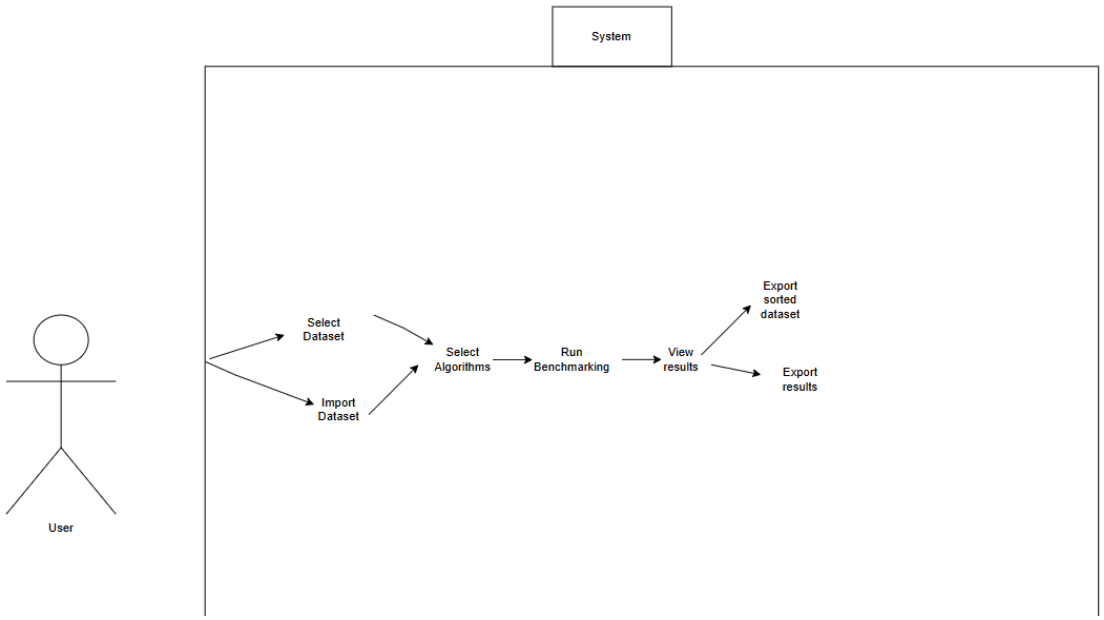


UML diagram

UML diagram uploaded to a google drive link, as it was difficult to show everything in the screenshot clearly. Link to the google drive: uml diagram.



Use case diagram



Essential algorithms/Pseudocode

The Benchmark Engine is the core original algorithm designed for this IA. It ensures fair, repeatable, and accurate benchmarking by cloning datasets, timing execution, counting iterations, and packaging results into BenchmarkResult objects.

Pseudocode: Benchmark Engine Algorithm
procedure BEFE3_RunBenchmark(dataset, algorithmList, runs, searchKey):

```
if dataset.size = 0 then
    raise "Dataset cannot be empty."
end if

if runs < 1 then
    raise "Number of runs must be ≥ 1."
end if

if length(algorithmList) < 2 then
    raise "At least two algorithms must be selected."
end if

results ← empty list

// Create dataset fingerprint to ensure fairness
originalHash ← hash(dataset.data)

for each alg in algorithmList do
```

```
timeList ← empty list
iterationList ← empty list

// Warm-up run to stabilise JVM behaviour
warmUpClone ← deepClone(dataset.data)
alg.run(warmUpClone)

for r from 1 to runs do

    cloned ← deepClone(dataset.data)

    // Verify integrity and fairness
    if hash(cloned) ≠ originalHash then
        raise "Dataset clone mismatch detected."
    end if

    alg.resetIterations()

    start ← nanoTime()
    try:
        if alg.type = SORT then
            alg.run(cloned)
        else:
            alg.run(cloned, searchKey)
    catch Error:
        print "Algorithm failed unexpectedly; skipping run."
        continue
    end try
    finish ← nanoTime()

    timeTaken ← finish - start

    // Adaptive anomaly detection
    if timeTaken > DYNAMIC_THRESHOLD_FACTOR * median(timeList) then
        print "Anomaly detected for " + alg.name + " in run " + r
        continue
    end if

    timeList.append(timeTaken)
    iterationList.append(alg.getIterations())

end for

// Statistical filtering of outliers
filteredTimes ← removeOutliers(timeList)
avgTime ← mean(filteredTimes)
medTime ← median(filteredTimes)
varTime ← variance(filteredTimes)
```

```
    sdTime ← sqrt(varTime)

    avgIterations ← mean(iterationList)
    sdIterations ← sqrt(variance(iterationList))

    result ← new BenchmarkResult(
                name = alg.name,
                avgTime = avgTime,
                median = medTime,
                stdDeviation = sdTime,
                avgIterations = avgIterations,
                iterationStdDev = sdIterations,
                complexity = alg.theoreticalComplexity
            )

    results.append(result)

  end for

  return results
```

Variable definitions used for all the Pseudocode

All pseudocode is either written on my own, or adapted from a source, which has been referenced. I did make changes after taking it from the source, to make this suitable for my IA.

```
ARRAY = list or array being sorted or searched
NUMBER_OF_ELEMENTS = number of items in ARRAY

I = outer loop index
J = inner loop index
K = secondary index for three-level looping
TEMP = temporary value used during swapping

KEY = value being inserted (Insertion Sort) or searched for (Search algorithms)

LOW = lowest index of current search range (Binary, Interpolation)
HIGH = highest index of current search range
MID = middle index in Binary Search
POS = estimated position (Interpolation Search)

STEP = block size (Jump Search)

CURRENT_SMALLEST_POSITION = index where smallest element should be placed (Selection Sort)
SMALLEST_ELEMENT = smallest element found in unsorted part

LEFT = left sub-array (Merge Sort)
RIGHT = right sub-array
MERGED = merged sorted array

PIVOT = value used to partition Quick Sort array

HEAP_SIZE = number of elements in heap
ROOT = index of root element during heapify

BUCKET = array used in Counting Sort
MAX_VALUE = maximum integer in ARRAY

ITERATIONS = number of comparisons / operations executed
```

Bubble sort:(Cormen et al., 2009)

The non optimized bubble sort algorithm is used, so the Big O remains constant at O(n^2)

```
FUNCTION BUBBLE_SORT(ARRAY) RETURNS INTEGER

    NUMBER_OF_ELEMENTS ← LENGTH(ARRAY)
    ITERATIONS ← 0

    FOR I ← 1 TO NUMBER_OF_ELEMENTS − 1 DO
        FOR J ← 1 TO NUMBER_OF_ELEMENTS − I DO
            ITERATIONS ← ITERATIONS + 1

            IF ARRAY[J] > ARRAY[J + 1] THEN
                TEMP ← ARRAY[J]
                ARRAY[J] ← ARRAY[J + 1]
                ARRAY[J + 1] ← TEMP
            END IF

        END FOR
    END FOR

    RETURN ITERATIONS
END FUNCTION
```

Selection Sort: (Y. Daniel Liang)

```
FUNCTION SELECTION_SORT(ARRAY) RETURNS INTEGER
    NUMBER_OF_ELEMENTS ← LENGTH(ARRAY)
    ITERATIONS ← 0

    FOR I ← 1 TO NUMBER_OF_ELEMENTS − 1 DO
        CURRENT_SMALLEST_POSITION ← I

        FOR J ← I + 1 TO NUMBER_OF_ELEMENTS DO
            ITERATIONS ← ITERATIONS + 1

            IF ARRAY[J] < ARRAY[CURRENT_SMALLEST_POSITION] THEN
                CURRENT_SMALLEST_POSITION ← J
            END IF

        END FOR

        TEMP ← ARRAY[I]
        ARRAY[I] ← ARRAY[CURRENT_SMALLEST_POSITION]
        ARRAY[CURRENT_SMALLEST_POSITION] ← TEMP

    END FOR

    RETURN ITERATIONS
END FUNCTION
```

Insertion sort:(Valiente)

```
FUNCTION INSERTION_SORT(ARRAY) RETURNS INTEGER
    NUMBER_OF_ELEMENTS ← LENGTH(ARRAY)
    ITERATIONS ← 0

    FOR I ← 2 TO NUMBER_OF_ELEMENTS DO
        KEY ← ARRAY[I]
        J ← I − 1

        WHILE J ≥ 1 AND ARRAY[J] > KEY DO
            ITERATIONS ← ITERATIONS + 1
            ARRAY[J + 1] ← ARRAY[J]
            J ← J − 1
        END WHILE

        ARRAY[J + 1] ← KEY
    END FOR

    RETURN ITERATIONS
END FUNCTION
```

Merge sort:(Cormen)

```
FUNCTION MERGE_SORT(ARRAY) RETURNS INTEGER
    NUMBER_OF_ELEMENTS ← LENGTH(ARRAY)

    IF NUMBER_OF_ELEMENTS ≤ 1 THEN
        RETURN 0
    END IF

    MID ← FLOOR(NUMBER_OF_ELEMENTS / 2)

    LEFT ← ARRAY[1..MID]
    RIGHT ← ARRAY[MID+1..NUMBER_OF_ELEMENTS]

    ITERATIONS ← MERGE_SORT(LEFT) + MERGE_SORT(RIGHT)

    MERGED, COUNT ← MERGE(LEFT, RIGHT)
    ITERATIONS ← ITERATIONS + COUNT

    ARRAY ← MERGED

    RETURN ITERATIONS
END FUNCTION
```

```
Merge function:
FUNCTION MERGE(LEFT, RIGHT) RETURNS (ARRAY, INTEGER)
    ITERATIONS ← 0
    MERGED ← EMPTY ARRAY
    I ← 1
    J ← 1

    WHILE I ≤ LENGTH(LEFT) AND J ≤ LENGTH(RIGHT) DO
        ITERATIONS ← ITERATIONS + 1

        IF LEFT[I] ≤ RIGHT[J] THEN
            APPEND LEFT[I] TO MERGED
            I ← I + 1
        ELSE
            APPEND RIGHT[J] TO MERGED
            J ← J + 1
        END IF
    END WHILE

    WHILE I ≤ LENGTH(LEFT) DO
        APPEND LEFT[I] TO MERGED
        I ← I + 1
    END WHILE

    WHILE J ≤ LENGTH(RIGHT) DO
        APPEND RIGHT[J] TO MERGED
        J ← J + 1
    END WHILE

    RETURN (MERGED, ITERATIONS)
END FUNCTION
```

Quick sort:(Hoare)

```
FUNCTION QUICK_SORT(ARRAY, LOW, HIGH) RETURNS INTEGER
    ITERATIONS ← 0

    IF LOW < HIGH THEN
        PIVOT, COUNT ← PARTITION(ARRAY, LOW, HIGH)
        ITERATIONS ← ITERATIONS + COUNT

        ITERATIONS ← ITERATIONS + QUICK_SORT(ARRAY, LOW, PIVOT − 1)
        ITERATIONS ← ITERATIONS + QUICK_SORT(ARRAY, PIVOT + 1, HIGH)
    END IF

    RETURN ITERATIONS
END FUNCTION
```

```
Partition functioning:
FUNCTION PARTITION(ARRAY, LOW, HIGH) RETURNS (INTEGER, INTEGER)
  PIVOT ← ARRAY[HIGH]
  ITERATIONS ← 0
  I ← LOW − 1

  FOR J ← LOW TO HIGH − 1 DO
    ITERATIONS ← ITERATIONS + 1

    IF ARRAY[J] ≤ PIVOT THEN
      I ← I + 1
      TEMP ← ARRAY[I]
      ARRAY[I] ← ARRAY[J]
      ARRAY[J] ← TEMP
    END IF

  END FOR

  TEMP ← ARRAY[I + 1]
  ARRAY[I + 1] ← ARRAY[HIGH]
  ARRAY[HIGH] ← TEMP

  RETURN (I + 1, ITERATIONS)
END FUNCTION
```

Heap sort: (Mark Allen Weiss)

```
FUNCTION HEAP_SORT(ARRAY) RETURNS INTEGER
    NUMBER_OF_ELEMENTS ← LENGTH(ARRAY)
    ITERATIONS ← 0
    HEAP_SIZE ← NUMBER_OF_ELEMENTS

    FOR I ← FLOOR(HEAP_SIZE / 2) DOWNTO 1 DO
        ITERATIONS ← ITERATIONS + HEAPIFY(ARRAY, HEAP_SIZE, I)
    END FOR

    FOR I ← HEAP_SIZE DOWNTO 2 DO
        TEMP ← ARRAY[1]
        ARRAY[1] ← ARRAY[I]
        ARRAY[I] ← TEMP

        HEAP_SIZE ← HEAP_SIZE − 1
        ITERATIONS ← ITERATIONS + HEAPIFY(ARRAY, HEAP_SIZE, 1)
    END FOR

    RETURN ITERATIONS
END FUNCTION
```

```
Heapify:
FUNCTION HEAPIFY(ARRAY, HEAP_SIZE, ROOT) RETURNS INTEGER
    ITERATIONS ← 0
    L ← ROOT * 2
    R ← ROOT * 2 + 1
    LARGEST ← ROOT

    IF L ≤ HEAP_SIZE AND ARRAY[L] > ARRAY[LARGEST] THEN
        LARGEST ← L
    END IF

    IF R ≤ HEAP_SIZE AND ARRAY[R] > ARRAY[LARGEST] THEN
        LARGEST ← R
    END IF

    ITERATIONS ← ITERATIONS + 2

    IF LARGEST ≠ ROOT THEN
        TEMP ← ARRAY[ROOT]
        ARRAY[ROOT] ← ARRAY[LARGEST]
        ARRAY[LARGEST] ← TEMP

        ITERATIONS ← ITERATIONS + HEAPIFY(ARRAY, HEAP_SIZE, LARGEST)
    END IF

    RETURN ITERATIONS
END FUNCTION
```

Shell sort: (Pratt)

```
FUNCTION SHELL_SORT(ARRAY) RETURNS INTEGER
    NUMBER_OF_ELEMENTS ← LENGTH(ARRAY)
    ITERATIONS ← 0
    GAP ← FLOOR(NUMBER_OF_ELEMENTS / 2)

    WHILE GAP > 0 DO
        FOR I ← GAP + 1 TO NUMBER_OF_ELEMENTS DO
            TEMP ← ARRAY[I]
            J ← I

            WHILE J > GAP AND ARRAY[J − GAP] > TEMP DO
                ITERATIONS ← ITERATIONS + 1
                ARRAY[J] ← ARRAY[J − GAP]
                J ← J − GAP
            END WHILE

            ARRAY[J] ← TEMP
        END FOR

        GAP ← FLOOR(GAP / 2)
    END WHILE

    RETURN ITERATIONS
END FUNCTION
```

Counting Sort: (Cormen, *Introduction to Algorithms*)

```
FUNCTION COUNTING_SORT(ARRAY) RETURNS INTEGER
    NUMBER_OF_ELEMENTS ← LENGTH(ARRAY)
    ITERATIONS ← 0

    MAX_VALUE ← MAXIMUM(ARRAY)
    BUCKET ← ARRAY OF SIZE MAX_VALUE INITIALISED TO 0

    FOR I ← 1 TO NUMBER_OF_ELEMENTS DO
        ITERATIONS ← ITERATIONS + 1
        BUCKET[ARRAY[I]] ← BUCKET[ARRAY[I]] + 1
    END FOR

    INDEX ← 1
    FOR K ← 1 TO MAX_VALUE DO
        WHILE BUCKET[K] > 0 DO
            ARRAY[INDEX] ← K
            BUCKET[K] ← BUCKET[K] - 1
            INDEX ← INDEX + 1
        END WHILE
    END FOR

    RETURN ITERATIONS
END FUNCTION
```

Linear Search

```
FUNCTION LINEAR_SEARCH(ARRAY, KEY) RETURNS INTEGER
    NUMBER_OF_ELEMENTS ← LENGTH(ARRAY)
    ITERATIONS ← 0

    FOR I ← 1 TO NUMBER_OF_ELEMENTS DO
        ITERATIONS ← ITERATIONS + 1

        IF ARRAY[I] = KEY THEN
            RETURN ITERATIONS
        END IF
    END FOR

    RETURN ITERATIONS
END FUNCTION
```

Binary search

```
FUNCTION BINARY_SEARCH(ARRAY, KEY) RETURNS INTEGER
    LOW ← 1
    HIGH ← LENGTH(ARRAY)
    ITERATIONS ← 0

    WHILE LOW ≤ HIGH DO
        MID ← FLOOR((LOW + HIGH) / 2)
        ITERATIONS ← ITERATIONS + 1

        IF ARRAY[MID] = KEY THEN
            RETURN ITERATIONS
        ELSE IF ARRAY[MID] < KEY THEN
            LOW ← MID + 1
        ELSE
            HIGH ← MID − 1
        END IF
    END WHILE

    RETURN ITERATIONS
END FUNCTION
```

Jump Search: (Cormen et al.)

```
FUNCTION JUMP_SEARCH(ARRAY, KEY) RETURNS INTEGER
    NUMBER_OF_ELEMENTS ← LENGTH(ARRAY)
    STEP ← FLOOR(SQRT(NUMBER_OF_ELEMENTS))
    ITERATIONS ← 0

    INDEX ← 1

    WHILE INDEX ≤ NUMBER_OF_ELEMENTS AND ARRAY[INDEX] < KEY DO
        ITERATIONS ← ITERATIONS + 1
        INDEX ← INDEX + STEP
    END WHILE

    INDEX ← INDEX − STEP

    WHILE INDEX ≤ NUMBER_OF_ELEMENTS AND ARRAY[INDEX] ≤ KEY DO
        ITERATIONS ← ITERATIONS + 1

        IF ARRAY[INDEX] = KEY THEN
            RETURN ITERATIONS
        END IF

        INDEX ← INDEX + 1
    END WHILE

    RETURN ITERATIONS
END FUNCTION
```

Interpolation search: (Knuth)

```
FUNCTION INTERPOLATION_SEARCH(ARRAY, KEY) RETURNS INTEGER
    LOW ← 1
    HIGH ← LENGTH(ARRAY)
    ITERATIONS ← 0

    WHILE LOW ≤ HIGH AND KEY ≥ ARRAY[LOW] AND KEY ≤ ARRAY[HIGH] DO
        POS ← LOW + ((KEY - ARRAY[LOW]) * (HIGH - LOW)) /
                    (ARRAY[HIGH] - ARRAY[LOW])

        POS ← FLOOR(POS)
        ITERATIONS ← ITERATIONS + 1

        IF ARRAY[POS] = KEY THEN
            RETURN ITERATIONS
        ELSE IF ARRAY[POS] < KEY THEN
            LOW ← POS + 1
        ELSE
            HIGH ← POS - 1
        END IF
    END WHILE

    RETURN ITERATIONS
END FUNCTION
```

Menu input validation

```
function readMenuChoice(min, max):

    valid ← false
    choice ← 0

    while valid = false do

        input ← readLine()

        if isInteger(input) = false then
            print "Invalid input. Please enter a number."

        else
            choice ← integer(input)

            if choice < min or choice > max then
                print "Choice out of range. Try again."
            else
                valid ← true
            end if

        end if
```

```
    end while

    return choice
```

CSV loading and validation

```
procedure loadFromCSV(path):

    if fileDoesNotExist(path) then
        print "Error: file not found."
        return

    list ← empty array

    open file at path

    while not end of file do
        token ← next value

        if token is empty then
            continue

        else if isInteger(token) = false then
            print "Error: non-integer value in file."
            close file
            return

        end if

        list.append(integer(token))

    end while

    close file

    if length(list) = 0 then
        print "Error: file contained no valid integers."
        return

    data ← list
    size ← length(list)
```

Dataset calculation

```
procedure ADCA_Classify(dataset):

    n ← dataset.size
    if n < 2 then
```

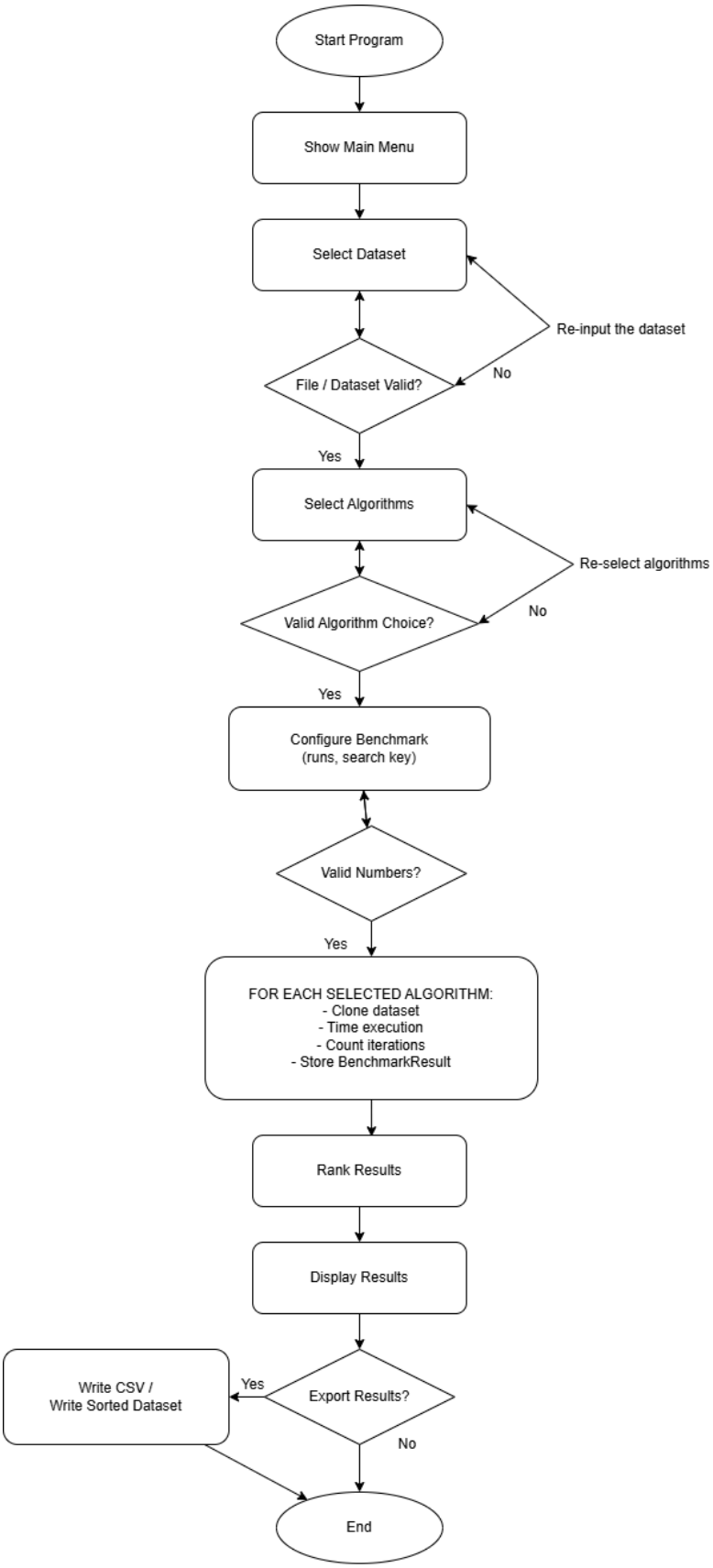
```
    return "insufficient data"
end if

inc ← 0
dec ← 0
inversions ← 0
breakpoints ← 0
runLength ← 1
maxRun ← 1

for i from 0 to n - 2 do

    if dataset[i] ≤ dataset[i+1] then
        inc ← inc + 1
    end if

    if dataset[i] ≥ dataset[i+1] then
        dec ← dec + 1
    end if

    if dataset[i] > dataset[i+1] then
        inversions ← inversions + 1
    end if

    if (dataset[i] > dataset[i+1]) then
        breakpoints ← breakpoints + 1
        maxRun ← max(maxRun, runLength)
        runLength ← 1
    else
        runLength ← runLength + 1
    end if

end for

sortednessScore ← inc / (n - 1)
descendingScore ← dec / (n - 1)
inversionRatio ← inversions / (n - 1)
runStability ← maxRun / n

if sortednessScore > 0.95 and breakpoints = 0 then
    return "sorted ascending"

else if descendingScore > 0.95 and breakpoints = 0 then
    return "sorted descending"

else if sortednessScore > 0.70 and runStability > 0.50 then
    return "nearly sorted"
```

```
    else
        return "random"
    end if
```

Export result to CSV

```
procedure exportResultsCSV(results, dataset, path):

    open file at path

    writeLine("datasetName,datasetType,algorithmName,time,comparisons,complexity")

    for each r in results do
        line ← dataset.name + "," +
               dataset.type + "," +
               r.algorithmName + "," +
               r.time + "," +
               r.comparisons + "," +
               r.complexity
        writeLine(line)
    end for

    close file
```

Flowcharts



Flowchart showing how the product is supposed to work.

Test Plan

Field	Test Type	Type of validation	Input data	Expected output	Pass Fail
Preset dataset choice	Normal	Range check	1	Accept value	Random dataset loads; name, size, sample values printed
	Extreme	Range check	Highest valid option (eg - 3)	Reject value and reinput	Correspondi ng preset dataset loads correctly.
	Abnormal	Range + type	abc	Reject value and reinput	Invalid input. Please enter a number. and re-prompt

Field	Test Type	Type of validation	Input data	Expected output	Pass Fail
CSV File path	Normal	Presence	rainfall.csv	Accept value	The content from the file will be extracted
CSV content	Extreme	Type check	1,2,x,5	Reject value and reinput	“Error: non-integer value in file.”; dataset not loaded.
CSV content	Abnormal	Logical	Empty file	Reject value and reinput	Invalid input. Please enter a number. and re-prompt

Field	Test Type	Type of validation	Input data	Expected output	Pass Fail
Algorithm selection	Normal	Range check	1 (Bubble), 4(Merge), 5(Quick)	Accept value	The content from the file will be extracted

Algorithm selection	Extreme	Logical	Only one algorithm	Reject value and reinput	“Please select at least two algorithms.” and re-prompt.
Algorithm selection	Abnormal	Logical	rainfall.csv	Reject value and reinput	“Please select at least two algorithms.” and re-prompt.

Field	Test Type	Type of validation	Input data	Expected output	Pass Fail
Dataset classification	Normal	Logical	nearly sorted dataset	Accept value	Classificatio n printed as “nearly sorted”.
Timing measuremen t	Normal	-	QuickSort on Random100	Accept value	Comparison count > 0 and repeatable for identical runs.
Big-O summary	Normal	Presence	Bubble + Quick on same dataset	Accept value	Summary text printed comparing $O(n^2)$ vs $O(n \log n)$ and linking to measured times.
Ranking table	Normal	Logical	Bubble, Insertion, Quick	Accept value	Table sorted from fastest to slowest algorithm.
Export results CSV	Normal	Format + presence	valid directory path	Accept value	CSV file created, with the correct header, one row per algorithm.
Export sorted dataset	Normal	Format + presence	valid directory path	Accept value	CSV file created with sorted

					integer values

Data dictionary

Attribute	Data type	Modifier	Comment
Dataset class			
data	int [ ]	private	The actual list of integers used in benchmarks.
name	String	private	Human-readable name, e.g. “Random100”, “CSV – marks.csv”.
type	String	private	RANDOM, NEARLY_SORTED, REVERSE_SORTED, CUSTOM_CSV.
size	int	private	Number of elements
DatasetManager Class			
presetDatasets	List <Dataset>	private	Stores predefined datasets for the user to select and load.
DatasetAnalyzer Class			
This class does not store data. It provides analysis utility methods (classification and sortedness).			
Algorithm: abstract class/interface: Common type for all algorithms (sort and search).			
algorithmName	String	protected	Name displayed in results, e.g. “Quick Sort”
algorithmType	enum	protected	Sorting or Searching
theoreticalComplexity	String	protected	Big O notation, for example O(n log n)
iterations	int	protected	Total comparisons/operations performed.
SortingAlgorithm (abstract)			

No additional attributes, inherits all properties from Algorithm			
SortingAlgorithm (abstract)			
No additional attributes, inherits all properties from Algorithm			
BenchmarkResult class			
Algorithm name	String	private	Name of the algorithm used in the benchmark.
execTimeNs	long	private	Exact time in nanoseconds.
iterations	int	private	Total comparisons/operations performed.
keyUsed	int	private	Search key (only for search algorithms)
BenchmarkEngine class			
result	List <Benchmark Result>	private	Stores all benchmark results for ranking/output
BenchmarkConfig Class			
dataset		private	Dataset chosen for the benchmark
algorithms	List <algorithm>	private	All selected algorithms to benchmark
searchKey	Integer	private	Search key entered by the user (null for sort-only runs).
AlgorithmBenchmarkingSystem (Main Controller)			
datasetManager	DatasetManager	private	Handles loading, cloning and retrieving datasets.
benchmarkEngine	benchmarkEngine	private	Runs sorting/searching benchmarks
inputValidator	inputValidator	private	Validates all user input

resultExporter	resultExporter	private	Exports benchmark results and sorted datasets

Appendix

Appendix A

Client: *Mr Ismith, IB Computer Science Teacher, Overseas Family School (OFS)*

Interviewer: *Tanish Gupta*

Q1. Could you please introduce yourself and your role at your school?

A:
“My name is Mr Ismith, and I am an IB Computer Science teacher at Overseas Family School. I teach both SL and HL Computer Science, and part of my role involves helping students understand algorithm design, data structures, and efficiency analysis within real programming environments.”

Q2. In your role, what kinds of datasets or data structures do you typically work with?

A:
“We most often work with integer arrays, string arrays, and data imported from simple CSV files. Students frequently experiment with small- to medium-sized datasets when exploring algorithms such as searches, sorts, recursion, and basic data structure manipulation.”

Q3. Currently, how do you or your students compare different algorithms for these datasets?

A:
“At the moment, we generally compare algorithms using individual Java programs created by students, or through theoretical analysis using Big-O notation. If we want empirical results, students normally add manual timing code into each algorithm and then observe the printed output in the console. It’s not a standardised process.”

Q4. What limitation does this create in your teaching or workflow?

A:
“The biggest issue is that students can’t easily compare algorithms under the exact same conditions. Each student writes their own timing logic, and small inconsistencies can make comparisons unreliable. It also takes a lot of time to set up fair tests, especially when switching between datasets with different characteristics. Because of this, students often don’t truly understand how input patterns affect performance.”

Q5. Why would an automated benchmarking system be useful?

A:
“An automated system would allow students to instantly benchmark multiple algorithms on the same dataset with consistent and objective timing logic. This would make demonstrations much clearer and would help connect theoretical complexity to real-world runtime behaviour. It would save time and provide a far more authentic learning experience.”

Q6. Broadly speaking, what would you want such a system to do?

A:
“I’d want it to load a dataset, allow me to choose a set of algorithms, run all of them under the same conditions, and then present their performance results clearly. Ideally, it should classify the dataset—whether it is random, sorted, nearly sorted—and run each algorithm once on a cloned version of that data so the comparison is fair.”

Q7. Would you want the system to allow: selecting algorithms, selecting datasets, loading CSV data, and running live comparisons?

A:
“Yes, definitely. Those features are essential. The more flexibility the user has, the more valuable the tool becomes. Teachers and students should be able to mix and match datasets and algorithms freely, including importing their own CSV files.”

Q8. Which algorithms should the tool support at minimum?

A:
“At the very least, it should include the core IB syllabus algorithms: Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, and Heap Sort. For searching, Linear Search and Binary Search would be the minimum. These are required in the IB course and would immediately make the tool useful for classroom learning.”

Q9. Would it be valuable if the benchmarking logic was coded from scratch?

A:
“Yes, absolutely. From an educational perspective, building your own benchmarking engine—handling cloning, timing, counting comparisons, and generating results—shows genuine algorithmic thinking and is much stronger than using built-in libraries. It would also make the IA more original and demonstrate a deeper understanding of computational processes.”

Q10. How should the results be displayed?

A:
“A simple, clear table is ideal. It should include the algorithm name, the execution time, the number of

comparisons, and the Big-O complexity. Sorting the results from fastest to slowest would make it very easy to interpret."

Q11. Would you want the results to be exportable?

A:
"Yes. Exporting to a CSV file would be extremely useful because I could load the results into Excel or Google Sheets to generate graphs or share them with my students. Exporting the sorted dataset would also be beneficial."

Q12. What internal data structures should the program use?

A:
"Arrays or ArrayLists make the most sense for storing datasets. For the benchmarking results, using ArrayLists or a dictionary-style structure would be appropriate. It should follow object-oriented principles so everything is modular and easy to extend."

Q13. Should the user have full control over which algorithms and datasets to test?

A:
"Yes, full control is important. Users should be able to choose any combination of algorithms and any dataset they want. The system shouldn't impose restrictions; flexibility is key."

Q14. What system will this run on?

A:
"A standard school laptop running Java is perfectly fine. A console-based interface is completely acceptable because the focus of the tool is on performance comparison and algorithmic understanding, not visual design."

Q15. What are the essential success criteria for this product?

A:
"I would consider the product successful if it can:

1. Load built-in datasets and import external CSV datasets reliably.
2. Validate input and avoid crashes.
3. Allow users to select at least two algorithms.
4. Classify the dataset correctly.
5. Clone the dataset and run each algorithm once for a fair comparison.
6. Measure execution time accurately using nanosecond timing.
7. Count comparisons or key operations.
8. Display a ranked results table with names, times, comparisons, and complexity.

9. Export results and the sorted dataset to CSV.
10. Be simple enough for a teacher or student to use without instructions."

Q16. Any optional future features?

A:
"In the future, you could add graphical visualisations, bar charts, or animations of the algorithms. More dataset patterns or larger benchmarks might also be nice. But these are optional; the core system you've planned is already more than sufficient."

Q17. Are you willing to act as the client for this IA and provide feedback when needed?

A:
"Yes, I'm happy to act as your client. I can provide feedback during development and test the product when you have a working version. It's an interesting idea and relevant to what we do in IB Computer Science."

Appendix B: Record of Tasks

Task number	Planned action	Planned outcome	Time estimated	Target completion date	Criterion
1	Get IA scenario and client approved by the teacher	Specific scenario and client approved	30 minutes	17/10/25	A (SDLC Analysis)
2	Scheduled first appointment with my client over WhatsApp and Google Calendar	Client describes his problem in cursory terms and agrees for a follow-up meeting	30 minutes	18/10/25	A (SDLC Analysis)
3	First face-to-face meeting with client (see Appendix)	Client specifies the problems he faces with current system	15 minutes	19/10/25	A (SDLC Analysis)

4	Ask follow-up questions to clarify details	Clearer understanding of client's requirements	20 minutes	20/10/25	A (SDLC Analysis)
5	Pitch proposed solution to CS advisor (teacher)	Teacher confirms feasibility and appropriateness of solution	20 minutes	21/10/25	A (SDLC Analysis)
6	Formulate success criteria based on client demands	Full list of measurable success criteria finalised	1 hour	22/10/25	A (Planning)
7	Brainstorm major system components	Initial overview of modules for decomposition diagram	45 minutes	13/11/25	B (Design Overview)
8	Create decomposition (top-down) diagram	Shows clear breakdown of system modules	45 minutes	13/11/25	B (Design Overview)
9	Draft first version of UML class diagram	Basic structure of classes and relationships created	1 hour	13/11/25	B (Design Overview)
10	Add abstract classes & inheritance to UML	Correct OOP architecture with polymorphism	45 minutes	14/11/25	B (Design Overview)
11	Add concrete sorting and searching algorithm	UML fully reflects final system functionality	30 minutes	14/11/25	B (Design Overview)

	subclasses to UML				
12	Create use-case diagram	Defines interaction between user and system	40 minutes	14/11/25	B (Design Overview)
13	Create defining diagram (Input – Processing – Output)	Shows all system I/O clearly	30 minutes	15/11/25	B (Design Overview)
14	Build full system flowchart / DFD	Complete visual representation of program flow	1 hour	15/11/25	B (Design Overview)
15	Produce data dictionary	Clear definitions for all attributes in UML classes	45 minutes	16/11/25	B (Design Overview)
16	Create test plan based on success criteria	Tests cover normal, boundary, and abnormal data	1 hour	16/11/25	B (Design Overview)
17	Finish the criteria A and B draft	Touch up everything, put the referencing	1 hour	16/11/25	A,B

Bibliography

Works Cited

Cormen, Al. *Introduction to Algorithms*. Cambridge, Mass., Mit Press, 2003.

---. *Introduction to Algorithms*. Cambridge, Mass., Mit Press, 2003.

Cormen, Thomas, et al. *Introduction to Algorithms*. Third ed., Cambridge, Mass., Mit Press, 2009.

Knuth, Donald E. *The Art of Computer Programming*. Addison-Wesley Professional, 24 Apr. 1998.

---. *The Art of Computer Programming*. Addison-Wesley Professional, 24 Apr. 1998.

“MacBook Specs | Technology Support Services.” *Nmu.edu*, 2024, it.nmu.edu/docs/macbook-specs.

Mark Allen Weiss. *Data Structures and Algorithm Analysis in Java*. Boston, Mass, Addison-Wesley, 2012.

Pratt, Vaughan R. *Shellsort and Sorting Networks*. Dissertations-G, 1979.

Valiente, Gabriel. *Algorithms on Trees and Graphs*. Springer Nature, 11 Oct. 2021.

Y. Daniel Liang. *Introduction to Java Programming, Brief Version*. 2020.