Technical Documentation: Arrays, Searching, Sorting, and Algorithm Analysis

Table of Contents:

1. **2D Arrays**
    a. Understanding 2D Array Storage
    b. Declaring and Initializing 2D arrays
    c. Modifying and Accessing elements in a 2D array
    d. 2D array of Objects
2. **Sorting and Searching Algorithms**
    a. TDD
    b. Sorting Algorithms
    c. Searching Algorithms
3. **Algorithm analysis and complexity**
    a. Linear Search Complexity and analysis
    b. Binary Search Complexity and analysis
    c. Comparing algorithms

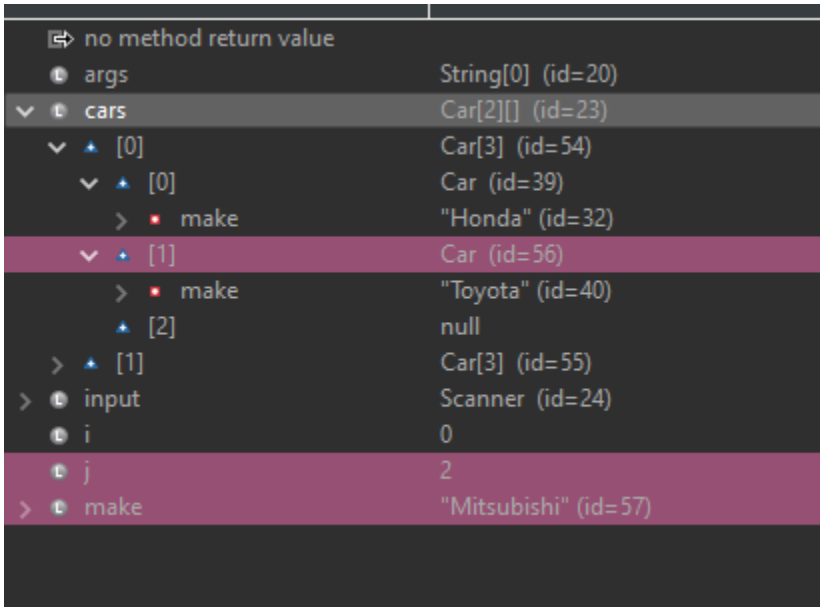
1. Exploring 2D Arrays

1.1 Understanding 2D Array Storage

A 2D array is stored in memory with row references pointing to separate column arrays on the heap. Example visualization:

int[][] numVals = new int[4][5];

Each row is stored separately in the heap, while the reference to the array resides in the stack as seen through Eclipse's debugger below.



1.2 Declaring and Initializing 2D Arrays

Syntax and Example:

dataType[][] arrayName;
int[][] numVals = new int[4][5];

Visual Representation:

Rows/Columns	Column 1(numVals[i][0])	Column 2(numVals[i][1])	Column 3(numVals[i][2])	Column 4(numVals[i][3])
Row 1(numVals[0][i])	0	0	0	0
Row 2(numVals[1][i])	0	0	0	0
Row 3(numVals[2][i])	0	0	0	0
Row 4(numVals[3][i])	0	0	0	0
Row 5(numVals[4][i])	0	0	0	0

1.3 Modifying and Accessing Elements in 2D Arrays

If we have a 2D array of integers, we can modify the elements by finding the slot where we would like to modify, like so:
numVals[2][1] = 6;
This will check the numVals row[2](the third row as it iterates from 0-2) and then checks column[1](The second column as it iterates from 0-1)

Iteration:

To iterate through your 2D array you would need a nested for loop as shown:

```
for(int i = 0; i < numVals.length; i++){
  for(int j = 0; j < numVals[i].length; j++){
    System.out.println(numVals[i][j]);
  }
}
```

What this does is iterate through each row and column to print out the values in the 2D array numVals.

1.4 2D Arrays of Objects

Using this example *Car* class, we can declare and initialize a 2D array that has the type Car and contains only car objects.

```
class Car{
```

```
    private String make;
    public Car() { this.make = "Unknown"; }
    public Car(String make) { this.make = make; }
    public void printMake() { System.out.print(this.make + " "); }
}
```

Declaring and initializing:

```
Car[][] cars = new Car[2][3];
for(int i =0; i < cars.length; i++){
    for(int j = 0; j < cars[i].length; j++){
        cars[i][j] = new Car("Toyota");
    }
}
```

2. Sorting and Searching Algorithms

2.1 Test-Driven Development (TDD) Approach

Test Driven Development(TDD) is a process of writing any program by creating test cases that the program can run into, then looking at the different conditions that are possible with their expected outcomes, and finally creating an algorithm or pseudocode that one can follow to create the minimum amount of code that is needed to run the program for that case.

Example Test Cases for Bubble Sort

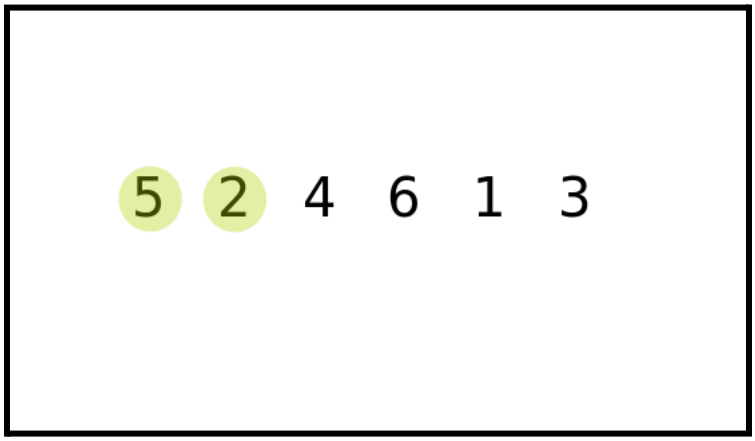
Precondition	Postcondition
[1,4,6,2,8]	[1,2,4,6,8]
[9,8,7,6]	[6,7,8,9]

2.2 Sorting Algorithms

Bubble Sort

```
void bubbleSort(int arr[]) {
    for (int i = 0; i < arr.length - 1; i++) {
        for (int j = 0; j < arr.length - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
```

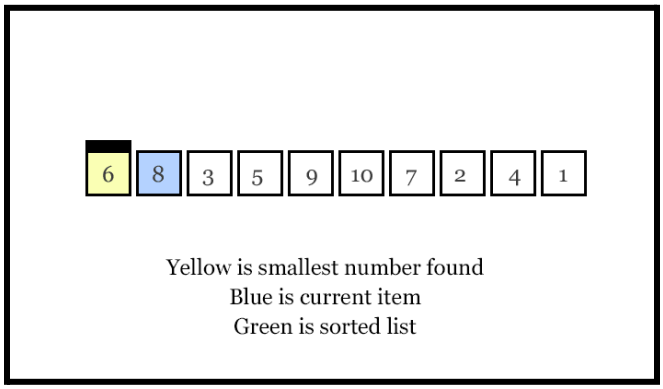
```
            arr[j + 1] = temp;
        }
    }
  }
}
```



Example of Bubble Sort ^

Selection Sort

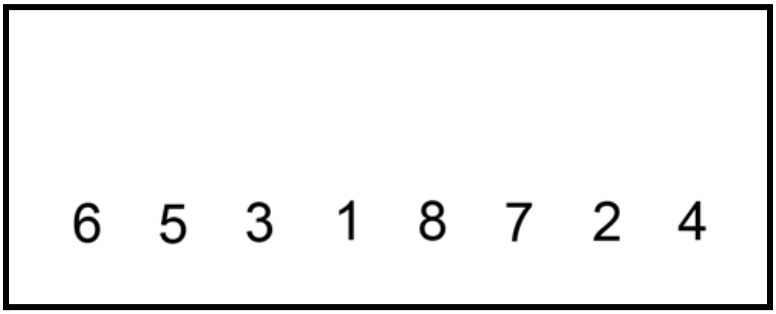
```
void selectionSort(int arr[]) {
  for (int i = 0; i < arr.length - 1; i++) {
    int minIndex = i;
    for (int j = i + 1; j < arr.length; j++) {
      if (arr[j] < arr[minIndex]) minIndex = j;
    }
    int temp = arr[minIndex];
    arr[minIndex] = arr[i];
    arr[i] = temp;
  }
}
```



Example of Selection sort ^

Insertion Sort

```
void insertionSort(int arr[]) {
    for (int i = 1; i < arr.length; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

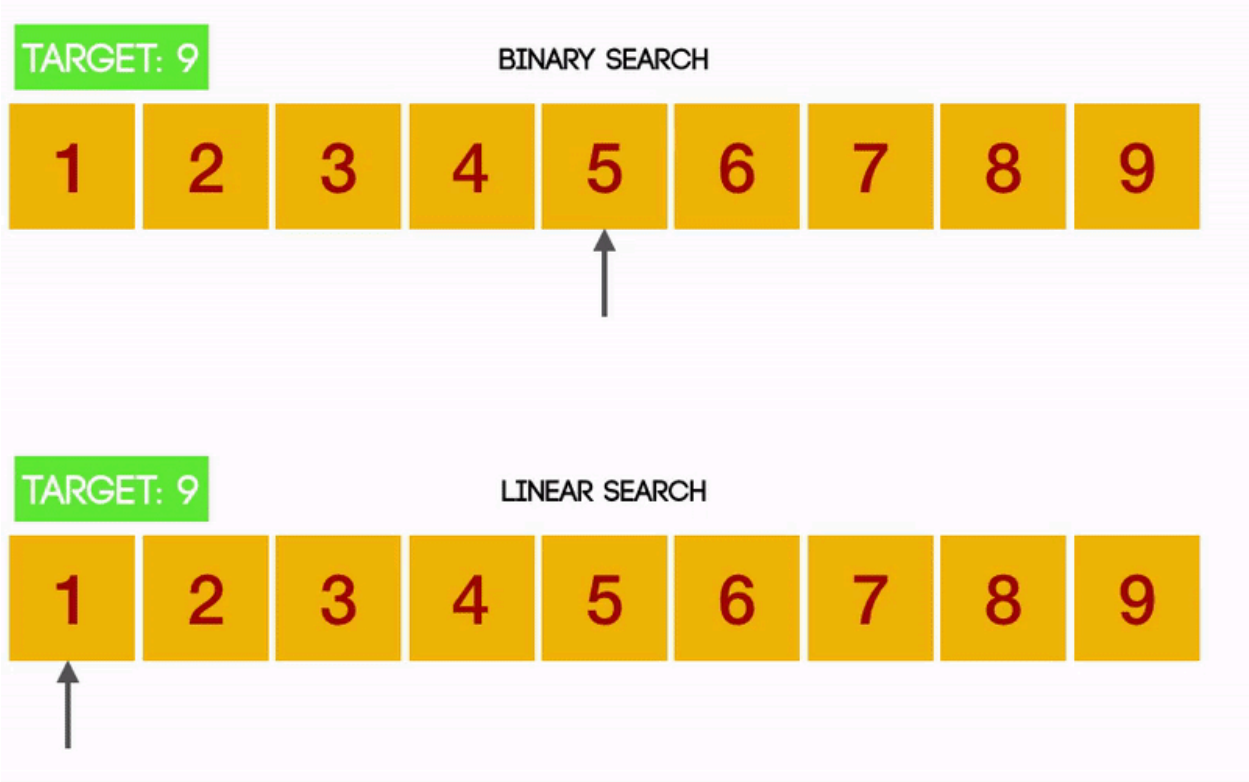


Insertion Sort example ^

2.3 Searching Algorithms

Linear Search

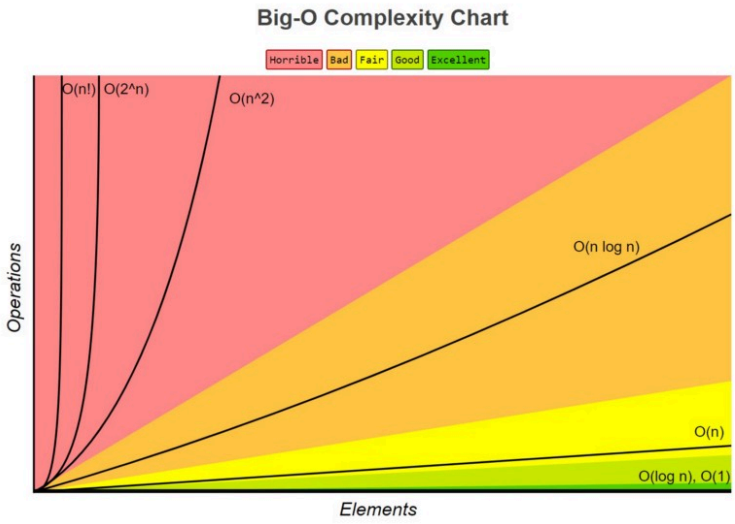
```
int linearSearch(int arr[], int key) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == key) return i;
    }
    return -1;
}
```



Binary Search

```
int binarySearch(int arr[], int key) {
    int left = 0, right = arr.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == key) return mid;
        if (arr[mid] < key) left = mid + 1;
        else right = mid - 1;
    }
    return -1;
}
```

3. Algorithm Complexity and Big-O Notation



3.1 Linear Search Complexity

- Worst-case: **O(N)**
- If array size = 1000, worst case comparisons = **1000**

3.2 Binary Search Complexity

- Worst-case: **O(log N)**
- If array size = 1000, worst case comparisons ≈ **10**

3.3 Comparing Search Algorithms

When looking at sorted arrays, we can say that Binary search is a better searching method when it comes to large sorted datasets, so generally it would be the best searching algorithm, however, for smaller sorted datasets, Linear search can be just as efficient if not more and require less comparisons to find the value. To determine the best method or algorithm to use, whether it be searching or sorting, it's best to look at the type of array you have, and use Big O notation to determine the quickest algorithm for the task.

Method	Best for
Linear Search	Small or unsorted arrays
Binary Search	Large sorted datasets