O-Notation & Algorithm Analysis

Winston Tsia, Nicole Mills-Dunning, Garrick Ngai

September 2023

1 Linear Search

Code by Garrick

```java

public class LinearSearch {
    // Declare the names array as a global variable
    static String[] names = {
        "Alice", "Bob", "Catherine", "David", "Eva", "Frank", "Grace", "Henry", "Irene", "Jack", "Karen", "Luke",
        "Mary", "Nathan", "Olivia", "Paul", "Quincy", "Rachel", "Samuel", "Tina", "Ulysses", "Victoria", "William",
        "Xander", "Yasmine", "Zachary",
        "Anna", "Benjamin", "Claire", "Daniel", "Emily", "Fiona", "George", "Hannah", "Isaac", "Julia", "Kevin", "Lily",
        "Michael", "Nora", "Oscar", "Penelope", "Quinn", "Ryan", "Sophia", "Thomas", "Uma", "Vincent", "Wendy", "Xavier",
        "Yara", "Zane", "name"
    };

    public static int caseInsensitiveSearch(String target) {
        for (int i = 0; i < names.length; i++) {
            if (names[i].equalsIgnoreCase(target)) {
                return i; // Return the index if the target name is found (case-insensitive)
            }
        }
        return -1; // Return -1 if the target name is not found
    }

    public static void main(String[] args) {
        Scanner scnr = new Scanner(System.in);

        System.out.print("Enter the target name: ");
        String target = scnr.nextLine();

        long startTime = System.nanoTime(); // Record the start time

        int index = caseInsensitiveSearch(target);

        long endTime = System.nanoTime(); // Record the end time
        double durationSeconds = (endTime - startTime) / 1e9; // Calculate the
        ↪  time taken in seconds

        if (index != -1) {
            System.out.printf("Found '%s' at index %d.\n", names[index], index);
        } else {
            System.out.printf("'%s' not found in the list.\n", target);
        }

        System.out.printf("Time taken: %.6f seconds (O(n) complexity)\n",
        ↪  durationSeconds);
        scnr.close();
    }
}
```

1.1 O-Notation

In the above code, a linear search is performed through the `names` array to find the key `name`, since this search algorithm iterates through each element of the array, the time it takes to find the target name is directly proportional to the number of elements in the array. This is classically denoted as:

$$O(n)$$

Where n represents the size of the input which is the `names` array. In the worst-case scenario, all elements are iterated through s.t. the last element of the array matches the key, resulting in $O(n)$ time complexity.

Since constant time operations are not considered for worst-case in O-notation, the constant time operations will be represented by c within the function. Acknowledging constant time operations:

$$f(n) = O(n) + c$$

$$O(f(n)) = O(n)$$

2 Recursive Binary Search

Code by Nicole

```java
public class RecursionBinarySearchLab {
        public static void main(String[] args) {

                String[] names = { "Alice", "Bob", "Catherine", "David", "Eva", "Frank", "Grace", "Henry", "Irene", "Jack",
                                "Karen", "Luke", "Mary", "Nathan", "Olivia", "Paul", "Quincy", "Rachel", "Samuel", "Tina", "Ulysses",
                                "Victoria", "William", "Xander", "Yasmine", "Zachary", "Anna", "Benjamin", "Claire", "Daniel", "Emily",
                                "Fiona", "George", "Hannah", "Isaac", "Julia", "Kevin", "Lily", "Michael", "Nora", "Oscar", "Penelope",
                                "Quinn", "Ryan", "Sophia", "Thomas", "Uma", "Vincent", "Wendy", "Xavier", "Yara", "Zane", "NAME" };

                Arrays.sort(names);

                System.out.println("----- RECURSIVE BINARY SEARCH -----");
                System.out.println();

                System.out.println("The names in the list: ");

                for (int i = 0; i < names.length; i++) {
                        System.out.println((names)[i] + " ");
                        names[i] = names[i].toLowerCase();
                }
                System.out.println();

                Scanner scnr = new Scanner(System.in);
                System.out.print("Enter an name to search for: ");
                String key = scnr.next().toLowerCase();

                int result = binarySearch(names, key);

                if (result == -1){
                        System.out.println("Element not present.");
                }
                else{
                        System.out.println("Element found at " + result + " index.");
                }
```

```java
		scnr.close();
	}

	/**
	 * Helper method to perform binary search with four parameters
	 *
	 * @param arr  list of elements to be searched
	 * @param word element to determine a match
	 * @return element
	 */

	public static int binarySearch(String[] arr, String word) {

		return binarySearch(arr, word, 0, arr.length - 1);
	}

	/**
	 * Method to perform recursion
	 *
	 * @param arr   list of elements to search
	 * @param word  element to match
	 * @param left  smallest value for index
	 * @param right largest value for index
	 * @return element
	 */

	public static int binarySearch(String[] arr, String word, int left, int
	↪  right) {


		if (left == right && !arr[left].equalsIgnoreCase(word)) {
			return -1;
		} else if (left == right && !arr[right].equalsIgnoreCase(word)) {
			return -1;
		}

		// Calculating middle point
		int middle = left + (right - left) / 2;

		int result = word.compareTo(arr[middle]);

		// Check if word is present at mid-point
		if (result == 0) {
			return middle;
		}

		// x on right side with subsearch from middle to right indexes.
		else if (result > 0) {
```

```
            return binarySearch(arr, word, middle + 1, right);
        }

        // x on left side with subsearch from left to middle indexes.
        else {
            return binarySearch(arr, word, left, middle - 1);
        }

    }

}
```

2.1 O-Notation

The above code implements a binary search algorithm to search for a target name in a sorted array of names. The time complexity of binary search is:

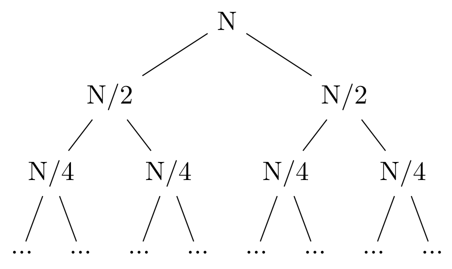
$$O(log\ n)$$

Initially, the data is modified to be lowercase, performing an $O(n)$ operation. The `binarySearch` method is called recursively, beginning on a sorted array, to search for the target name within a sub-array of the array. At each step, the algorithm divides the search range in half, which results in a logarithmic time complexity. The time complexity of this binary search is $O(log\ n)$, which is also an inverse function of exponentiation, making it much more efficient than a linear search $O(n)$ for larger arrays, especially for sorted data. However, because a linear operation is performed on the data, the dominant term in consideration is $O(N)$, meaning the time complexity is linear. Acknowledging constant time operations:

$$T(N) = O(N) + T(N/2) + c$$

$$O(T(N)) = O(N) + O(log\ N) = O(N)$$

2.2 Recursive Tree



The binary search algorithm splits the data in half, and calls the recursive function on the remaining half, forming the above recursive tree.

3 Sierpinski Triangle

Code by Winston

```java
// shortened for readability
public class SierpinskiTriangle {
    private static void drawSierpinski(int sideLength, int depth) {
        double[][] triangle = sierpinskiGenerator(sideLength, depth);
        sierpinskiDivider(triangle, depth);
        // print
    }

    private static double[][] sierpinskiDivider(double[][] inputTriangle, int
    ↪ depth) {
        if (depth == 0) {
            return inputTriangle;
        };

        double[][]triangleMidpoint = new double[4][];
        triangleMidpoint[0]= midpoint(inputTriangle[0], inputTriangle[1]);
        triangleMidpoint[1]= midpoint(inputTriangle[0], inputTriangle[2]);
        triangleMidpoint[2]= midpoint(inputTriangle[1], inputTriangle[2]);

        // substitute data into initial triangle's last empty array for segment
        ↪ removal w/ height and length
        inputTriangle[3][0] = triangleMidpoint[0][0];
        inputTriangle[3][1] = triangleMidpoint[1][0] - triangleMidpoint[0][0];

        double[][]subTriangle1 = new double[4][];
        subTriangle1[0] = inputTriangle[0];
        subTriangle1[1] = triangleMidpoint[1];
        subTriangle1[2] = triangleMidpoint[2];

        double[][]subTriangle2 = new double[4][];
        subTriangle2[0] = triangleMidpoint[0];
        subTriangle2[1] = inputTriangle[1];
        subTriangle2[2] = triangleMidpoint[1];

        double[][]subTriangle3 = new double[4][];
        subTriangle3[0] = triangleMidpoint[2];
        subTriangle3[1] = triangleMidpoint[1];
        subTriangle3[2] = inputTriangle[2];

        sierpinskiDivider(subTriangle1, depth - 1);
        sierpinskiDivider(subTriangle2, depth - 1);
        sierpinskiDivider(subTriangle3, depth - 1);

        return inputTriangle;
    };

    private static double[][] sierpinskiGenerator(int sideLength, int depth) {
        // Triangle formed from bottom left at origin (0,0), midpoint between left
        ↪  and top, top with x,y coord, ...
        double[][] baseTriangle = new double[depth*6][];
        baseTriangle[0] = new double[]{0, 0};
        baseTriangle[1] = new double[]{sideLength / 2.0, Math.sqrt(3) / 2.0 *
        ↪  sideLength};
        baseTriangle[2] = new double[]{sideLength, 0};
        baseTriangle[3] = new double[]{0, 0}; // initialize removal height and
        ↪  removal length as 0
        return baseTriangle;
    };

    private static double[] midpoint(double[] A, double[] B) {
        return new double[]{ Math.abs(A[0] + B[0]) / 2.0 , Math.abs(A[1] + B[1])
        ↪  / 2.0};
    };

    public static void main(String[] args) {
        int sideLength = 16;
        int depth = 2;
        drawSierpinski(sideLength, depth);
    }
}

```

3.1 O-Notation

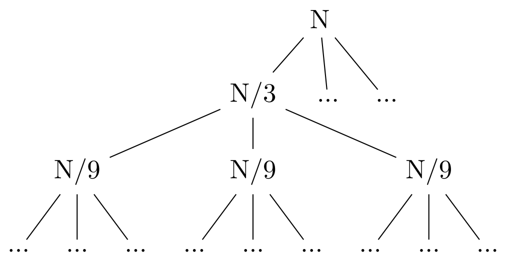
Because the Sierpinski Triangle is a fractal pattern that relies on depth to generate complexity, we can consider our input to be depth N. For a depth, of 0, we generate a single triangle. For a depth of 1, we generate a single triangle as well as 3 sub-triangles. For a depth of 2, we generate a total of 9 triangles. This pattern generally follows 3^n. The progression follows a sequence s.t.:

$$A_n = \{1, 3, 9, ...\}$$

Furthermore, an operation is done to remove the inner triangle, creating one additional operation. Therefore, a recursive Sierpinski Triangle has the following complexity for some depth N:

$$T(N) = O(3^N)$$

3.2 Recursive Tree



As the base triangle is subdivided into its smaller triangles, the division operation creates 3 further sub-triangles, recursively creating an exponential time complexity with each depth.

3.3 Discussion

Noticeably, the Sierpinski Triangle takes the most time in completing, at exponential complexity, which makes generating large Sierpinski Triangles computationally expensive in terms of both time and memory. As for the other two algorithms, since sorting on the data-set is considered, both the linear and binary search algorithms have the same O-notation of $O(n)$, which is only the case because sorting the data is considered.

When comparing the recursive trees of both the Sierpinski Triangle and the recursive binary search, we can note that the recursive calls are more numerous for the triangle, where depth creates exponential complexity that exceeds the speed of binary search.