Lab Assignment #1

Linear/Binary Search

Full Name: Shreyas Srinivasa
BlazerId: SSRINIVA

CS 303 Algorithms and Data Structures

Spring 2023

1. Problem Specification

The purpose of this project was to implement algorithms for linear search and binary search and measure their execution time with randomized datasets. This execution time was further used to study the connection between real time performance and theoretical complexity analyses.

2. Program Design

```java
public static int linearSearch(List<Integer> arr, Integer key) {
    for(int i = 0; i < arr.size(); i++) {
        if(arr.get(i).equals(key)) return i;
    }
    return -1;
}

public static int binarySearch(List<Integer> arr, int l, int u, Integer key) {
    if(l > u) return -1;
    int m = (l + u) / 2;
    if(arr.get(m) > key) {
        return binarySearch(arr, l, m - 1, key);
    } else if(arr.get(m) < key) {
        return binarySearch(arr, m + 1, u, key);
    } else return m;
}
```

First the algorithms of linear search and binary search were implemented in Java code

```java
    public static void driver1() {
        List<Integer> arr = Arrays.asList(22, 45, 65, 3, 1, 903, 43, 5);
        int n1 = 45, n2 = 75, n3 = 65, n4 = 5;
        Collections.sort(arr);
        System.out.println(arr);
        int linearResult1 = linearSearch(arr, n1);
        int linearResult2 = linearSearch(arr, n2);
        int binaryResult1 = binarySearch(arr, 0, arr.size() - 1, n3);
        int binaryResult2 = binarySearch(arr, 0, arr.size() - 1, n4);

        System.out.format("%d found at %d and %d found at %d\n", n1, linearResult1, n2,
linearResult2);
        System.out.format("%d found at %d and %d found at %d\n", n3, binaryResult1, n4,
binaryResult2);

    }
```

The algorithms were tested with a basic dataset and keys

```java
public static List<Integer> getNRandomNos(int n) {
    Random rand = new Random();
    List<Integer> numbers = new ArrayList<>();
    for(int i = 0; i < n; i++) {
        numbers.add(rand.nextInt(1000));
    }
    return numbers;
}

public static Map<Integer, List<Integer>> generate2NDatasets(int N) {
    Map<Integer, List<Integer>> datasets = new HashMap<>();
    for(int i = 4; i <= N; i++) {
        Integer n = (int)Math.pow(2, i);
        datasets.put(i, getNRandomNos(n));
    }
    return datasets;
}
```

Functions for generating random numbers and datasets with length of the order of 2^N were designed

```java
    public static List<Integer> readKeysFromFile() {
        List<Integer> keys = new ArrayList<Integer>();

        try (Scanner fileScanner = new Scanner(Paths.get("input_1000.txt"))) {
            while(fileScanner.hasNextLine()) {
                String line = fileScanner.nextLine();

keys.addAll(Arrays.asList(line.split(",")).stream().map(Integer::valueOf).collect(Collectors.toList(
)));
            }
            fileScanner.close();
        } catch (Exception e) {
            System.out.println("Error: " + e.toString());
        }
        return keys;
    }
```

Random keys between 1 to 1000 were read from a text file and stored in a data structure

```java
public static void driver2() {
    List<Integer> keys = readKeysFromFile();
    Map<Integer, List<Integer>> datasets = generate2NDatasets(20);
    Instant start, finish;

    for(Integer N = 4; N <= 20; N++) {
        List<Integer> dataset = datasets.get(N);
        long timeElapsed;

        System.out.format("2^%-5d", N);

        start = Instant.now();
        for(Integer key: keys) {
            Algorithms.linearSearch(dataset, key);
        }
        finish = Instant.now();
        timeElapsed = Duration.between(start, finish).toNanos() / 1000;
        System.out.format("%12d", timeElapsed);

        Collections.sort(dataset);
        start = Instant.now();
        for(Integer key: keys) {
            Algorithms.binarySearch(dataset, 0, dataset.size() - 1, key);
        }
        finish = Instant.now();
        timeElapsed = Duration.between(start, finish).toNanos() / 1000;
        System.out.format("%12d", timeElapsed);

        System.out.println();
    }
}
```

Function to calculate and print the average time taken in nanoseconds by each algorithm for each datasets was written. This produced our data for further study

3. **Output**

```
2^4             3065          661
2^5             1483          419
2^6             2372          433
2^7             4548          510
2^8             3250          521
2^9              674          543
2^10            1259          192
2^11            1455          419
2^12            1903          196
2^13            1870          554
2^14            2756          354
2^15            3754          255
2^16            2549          387
2^17            2952          364
2^18            4032          651
2^19            7254          482
2^20            7793          564
```

Size of Dataset	Linear Search(nanoseconds)	Binary Search(nanoseconds)
2^4	3605	661
2^5	1483	419
2^6	2372	433
2^7	4548	510
2^8	3250	521
2^9	674	543
2^10	1259	192
2^11	1455	419
2^12	1903	196
2^13	1870	554
2^14	2756	354
2^15	3754	255
2^16	2549	387
2^17	2952	364
2^18	4032	651
2^19	7254	482
2^20	7793	564

4. **Analysis and Conclusions**

A **linear search** scans each element of the array sequentially until the required element is identified. As a result, the worst time complexity is of the order O(n) since the time required to search an element is proportional to the number of the elements.

A **binary search** finds the required element in an array via middle element value comparison; hence, cutting the array length to be searched in each scan cycle by half. This leads to faster computation and reduces the worst time complexity to O(log n). However, the prerequisite for a binary search to work is that the array must be sorted.

The Time Complexity of an algorithm/code is **not** equal to the actual time required to execute a particular code, but the number of times a statement executes.

The **actual time required to execute code is machine-dependent** (whether you are using Pentium 1 or Pentium 5) and also it considers network load if your machine is in LAN/WAN.

Instead of measuring actual time required in executing each statement in the code,Time Complexity considers how many times each statement executes.

Therefore the relation between the measured execution time and the theoretical model of time complexity is not apparent. But there is a general trend towards a larger execution time with the increase in input length for linear search, whereas the execution time of binary search suffers minimal deviations comparatively and stays within a fixed range. This shows us that the time complexity analysis is a good approximation for real world machine specific performance.

This is the execution time after including the sorting time for binary search

Size of Dataset	Linear Search(nanoseconds)	Binary Search(nanoseconds)
2^4	3002	1401
2^5	1027	540
2^6	1765	703
2^7	3270	647
2^8	8104	943
2^9	876	1403
2^10	1316	2093
2^11	1620	2171
2^12	1182	3449
2^13	1947	5450
2^14	2561	12925
2^15	2561	26879
2^16	3363	62365
2^17	2731	103976
2^18	4332	182571
2^19	6090	142061
2^20	6842	269054

The sorting step, if using an efficient algorithm, will still have a time complexity of O(nlog(n)). Therefore we see a significant growth in execution time with increase in dataset size.

5. References

- https://dev.to/doabledanny/binary-search-javascript-plus-big-o-performance-explained-simply-3jbn
- https://docs.oracle.com/en/java/javase/17/docs/api/index.html
- https://www.geeksforgeeks.org/collections-sort-java-examples/
- https://www.geeksforgeeks.org/generating-random-numbers-in-java/
- https://www.scaler.com/topics/linear-search-and-binary-search/