Lab 3 Sort

Weichen Chai

Fall 2022

Introduction

This lab report seeks to answer the questions presented in "Searching for an array" for ID1021. In this assignment, the goal is to search for a key in an unsorted array, the task requires to look for the key after the array has been sorted, as it will be much easier.

Background

This task enhances our understanding in search algorithms. We will work with unsorted and sorted arrays, and analyse their differences. Binary search in arrays will also be experimented with, where the search method differs from the standard loop searching.

Tasks

Unsorted array

The first task in the assignment requires for us to benchmark the search function in an unsorted array, where the search function is just a simple piece of code which runs a loop through all the elements in the array until it is able to find the right one.

To find the relationship between size and time taken to sort through them, it is best to run the code in a loop a number of times to gather the average data, namely using the following code.

```
int[] Array = {50, 100,500 ,1000 , 5000,  10000};
int[] result;
int Loops = 100000;
double Total = 0;
.
.
.
```

```
    for (int j = 0; j < Loops; j++){
        result = Sorted(i);
```

The code that was presented allows the loop to be run for 100000 times. I have chosen this number because after run so many times, average found by dividing the total number of loops to the looped times will be extremely accurate, thus giving a clear visualization of how the overall trend should look like. The results give are the following:

Elements in array	Time in Nanoseconds
50	70.1
100	85.8
500	166.3
1000	258.6
5000	1187.8
10000	2402.7

Once these points are plotted in a graph, it is quite visible that they follow a linear pattern. As the size of the array increases, the number of time taken to find the key also increases.

Sorted stack

The assignment asks for our prediction without experiment for searching through an array with one million elements. When we take a look at the sorted array, we realize that it is mostly the same as the unsorted array, with the same search function. The main difference between the sorted and unsorted array is that the code for sorted array makes sure to place an element that is bigger than the current using the line

```
nxt += rnd.nextInt(10) + 1;
array[i] = nxt;
```

Other than that, the method for searching remains similar, where the function returns false if the number we are looking for has not been found, and returns true if the element has been found. We can conclude that it should take approximately the same amount of time as the unsorted search.

```
//optimized code for searched array
public static boolean search_sorted(int[] array, int key){
    for (int index = 0; index < array.length ; index++) {
        if (array[index] == key){
        return true;
```

```
        }


         //So that we know the element we looked for isn't there
        if(array[index] > key){
         return false;
        }

    }
    return false;
}
```

In order to predict the average amount of time taken to search through an array, we can try to find the line of regression of the data received from the unsorted array, which is ($y = 0.23x + 45.47$). Using the line of regression, we can substitute in one million and receive the result of around 230045.

Binary search

The binary search method utilizes much less resources and time than the sorted and unsorted search method, which simply uses a looping mechanic. This method first defines a first and last element, then jumps to the element in the middle of the first and last element. This method compares the key that it is looking for with the element in the middle, and will determine whether the element is bigger or smaller than the key. If the element is bigger, element + 1 will be defined as the new "last" element, if the element is smaller, element - 1 will act as the new "first" element. The process of jumping to the middle of the array continues until the key element looked for has been found.

The process can be visualized by the following code.

```
int first = 0;
int last = array.length - 1;
    while(true){
        int index = (last-first)+ first /2; //finding the middle between first and
        if (array[index] == key){
            return true;
            }
        if (array[index] < key && index < last){
             first = index + 1;
             continue;
            }
        if (array[index] > key&& index > first){
            last = index - 1;
            continue;
```

```
        }
.
.
.
```

Using the binary search method, we wish to find how long it takes to search through an array of one million elements, and estimate how long it would take to search through 16 million items. The data gathered is the following:

Elements in array	**Time in Nanoseconds**
1	56.2
50	66.1
100	83.9
500	116.3
1000	134.6
10000	183.8
100000	586.7
1000000	1121.5
16000000	1512.5

The data gathered indicates that the binary search algorithm works in a O(log(n)) manner, where the line of best fit is a logarithmic line. The pattern of such a line is that it will not experience major increases on the y Axis, which is the axis representing time in nanoseconds, after a certain point in the x axis (number of elements in an array). Searching through one million elements take around 1121.5 nanoseconds on average, as evaluated after 100000 runs. My prediction s that for 64 million elements in an array, the time taken to search through it will be similar to the time taken for 16 million, with only a slight increase in time. Getting the logarithmic regression line from the data received : y = 97.0694 + 71.2669log(x). so 64 million elements will result in 1705 nanoseconds

Even better

In the previous assignment, we were asked to search through the second array for all the items inside the first array. This tasks asks what would happen if we performed the same strategy for two sorted arrays. In the original assignment, the duplicate function had a complexity of *O(n2)*. In this task, the search function will be performed with binary search. As binary search have the complexity of O(Log(n)), we can conclude that searching through two duplicates will result in a complexity of O(n(Log(n))). By changing the binary search function, we get the following:

```
for (int j = 0; j < Keys.length; j++){
        New[] = binarySearch(Array, keys[j]);
}
```

A method described by the assignment instructions, when we compare the value from the two arrays, we can try to compare them from the size of the element. if the next element in the first array is bigger than the next element in the second array, then the second array will be incremented, vice versa. If the next element in the second array is equal or greater than the first array then we increment the first array. The complexity for this type of search array is going to be O(n). This method should take less and less time compared to the binary search technique as the amount of elements inside the array grows

Theoretically, comparing the initial duplicate function to binary search to the newly proposed algorithm that the tasks suggests, should result in O(n2) > O(n(Log(n)) > O(n)). The result of these three codes are the following

Elements	**Origin**	**Binary**	**New**
50	36	24	8
100	102	72	12
500	982	218	35
1000	1928	294	65
10000	45839	6192	596

Table 1: In microseconds