Sorted

Laasyasree Desu

Fall 2023

1 Introduction

This assignment involves finding the efficiency of different algorithms to search through sorted and unsorted arrays of integers. This is done by executing the methods, measuring the average time taken for it and determining the time complexity.

2 Unsorted

The first task involves evaluating the time to search through an unsorted array. The code for this was already given, the search algorithm goes through every element in the array and compares it to the key (integer being searched for). The additional parts were generating the unsorted array and implementing the benchmark to measure. The code for generating the array can be seen below:

```
private static int[] unsorted(int length) {
        Random number = new Random();
        int[] elements = new int[length];
        for (int i = 0; i < length ; i++) {
        elements[i] = number.next[length]; }
        return elements;
        }
```

3 Results

The results below show the time taken to search and unsorted array of various lengths. The measured time is an average of 10000 searches. It can be observed that the time taken to search with this algorithm is a linear function, since the it is proportional to the size of the array. Therefore the time complexity for this algorithm can be concluded to be O(n).

Array size	Time taken (ns)
100	17
200	34
400	70
800	128
1600	245

Table 1: Average time to search unsorted array.

4 Sorted

The second part of the task was to search through an array of sorted elements instead of unsorted. The code for a sorted array was already given, however modifications were made to search algorithm to make it more efficient (taking advantage of the array being sorted). This time if the element after the current element being compare to is larger than the key, then the search would be terminated.

```
public static boolean search_sorted(int[] array, int key) {
   for (int index = 0; index < array.length ; index++) {
       if (array[index] == key)
           return true;
       if(array[index+1]>key)
           return false;}
   return false;
}
```

5 Results

Array size	Time taken (ns)
100	19
200	32
400	53
800	70
1600	239

Table 2: Average time taken to search sorted array.

The results above show the average time taken to search through a sorted array of various lengths. The measured time ended up being similar to the time measured for the unsorted array, with slight improvements on some array lengths. This is because even though extra conditions have been put in to increase efficiency, the key searched for is randomized and therefore could potentially be found at the end of the array. This means the entire length of the array is still searched, and therefore would not make a difference in the time taken. Since this is similar to the previous algorithm, the time complexity for this is also O(n).

6 Binary Search

The second task is to implement a new searching algorithm known as the binary search. Unlike the previous ones, this algorithm starts in the middle of the array and moves a quarter of the way forward or backward. To be able to jump this way it also keeps track of the first and last elements. Taking advantage of the fact that the array is sorted, the algorithms first compares the key the middle value, if the key is less then the search field is modified to range from the first index to the middle index. If it is larger then the search field ranges from the middle index to the last index. This is repeated until the key is found. Additionally some conditions are added so that the range does not go out of bounds. An example of this code can be seen below:

```
if (array[mid] < key && middle < last) {
    first = middle+1;
    continue;
    }
```

7 Results

Array size	Time taken (ns)
100	34
200	40
400	46
800	53
1600	58

Table 3: Average time taken search sorted array using binary search.

The results show that this algorithm at first (for smaller array sizes) appears too be slower, but as the sizes increase there is no significant difference in the time taken (not proportional to the size of the array), making this the more efficient algorithm comparatively. Looking at the results it can be concluded that the time complexity for this algorithm is O(log n).

8 Duplicates

In this task the objective is to find an efficient algorithm to find duplicates in two sorted arrays. This was done by looping through one array while using the binary search algorithm on the second array. An example of this code can be seen below:

```
if (array[mid] < key && middle < last) {
    first = middle+1;
    continue;
    }
```

9 Results

Array size	Time taken (ns)
100	1042
200	2515
400	5775
800	27221
1600	68205

Table 4: Average time to find duplicates using binary search in sorted array.

The time complexity for this algorithm can be determined by splitting it into two different parts. The first part is the binary search algorithm that has time complexity O(log n). The second part is the for loop that runs the method n times. Therefore the complexity is O(n (log n)).

10 Duplicates - improved

In this task the algorithm to search for duplicates was modified to be even more efficient. This time instead of using the binary search algorithm, two pointers were used to keep track of the next elements in both arrays. If the next element is smaller in the first array, then the pointer for that moves forward. If the

current element in the second array is smaller, the same goes. If the elements from both arrays are equal the counter (keeping track of how many duplicates) is incremented and both pointers both move forward. An example of this code can be seen below:

```
if (arr1[id1]==arr2[id2]){
    id1++;
    idx2++;
    counter++;
    }
else if(arr1[id1] < arr2[id2])
        index1++;

else if (arr1[id1] > arr2[id2]){
        index2++;
```

11 Results

Array size	Time taken (ns)
100	305
200	632
400	1490
800	3743
1600	7567

Table 5: Average time taken to find duplicates using pointers in sorted array.

The results show the time to be proportional to the size of the array. Additionally, from the code it can be seen that the time depends on the length of the arrays, therefore the time complexity for this algorithm can be concluded to be O(n).