The Benefit of Sorted Data

Aditya Gupta

September 2022

Introduction

We have seen in previous assignments that the cost of searching for a key in an unsorted array is quite expensive. We will expand upon that idea through this assignment and see how having data that is presorted reduces the amount of time taken to find a key element significantly. This is because we have now obtained a specific structure within the array and can come up with more clever methods to find our key element. One such method is the Binary search algorithm, which we will be going over in this assignment and comparing against the linear search algorithm which is more of a brute force method that compares the key value against all elements inside the array. We will be using Java programming language to test and implement the task in this assignment.

Benchmarking Algorithms

In this section, we talk about the benchmarking results and conclusions drawn from the data obtained. We also compare the runtimes of different algorithms and make predictions for the runtimes of large datasets based on the trends that we observe for these algorithms.

Sequential Search Algorithm code

We set up a benchmark for the time taken to do a normal sequential search for any key element in a randomly generated array. We do this using the following pieces of code:

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

The logic of the algorithm is to simply compare the value of the key element against all values of the randomly generated array that is given to it as an argument. This is the approach that we have also used before and now we use this algorithm again just so that we can compare the binary search algorithm with it.

```
public static int[] sorted(int n) {
        Random rnd = new Random();
        int[] array = new int[n];
        int nxt = 0;
        for (int i = 0; i < n ; i++) {
            nxt += rnd.nextInt(10) + 1;
            array[i] = nxt;
        }
        return array;
    }
```

Here we manage to generate random arrays that are sorted, to begin with. Here we don't need to use any fancy sorting algorithm to get the sorted array as the function makes use of a clever trick. Here we generate a random integer and then add its value into an integer variable called `nxt` which we then set as the value for the first element. Now for each iteration after that, we keep adding random numbers generated using the `Random` class in java to this variable which we then use to set the values for the next position in the array.

The data we get upon benchmarking the above algorithm for different-sized arrays have been tabulated below:

Search Unsorted

Case	n	Time(in microseconds)	Ratio of time with the case 1 value as a reference
1	8000	3	1.00
2	16000	6	2.00
3	32000	14	4.67
4	64000	23	7.67
5	128000	42	14.00
6	256000	68	22.67
7	512000	121	40.33

From this, we can clearly see that the time taken for the sequential search algorithm given by the search-unsorted method increases linearly with the increase in the number of elements. This result was obtained upon doing the same algorithm for 10000 iterations.

From this, we can see that in the average case the time value increases by a ratio of 2-2.5 for the increase in the n value by a ratio of 2. This means that without running the above code we can predict that for a data set of 1 million elements the time taken to search for a key element should be in the range of 240 - 300 microseconds.

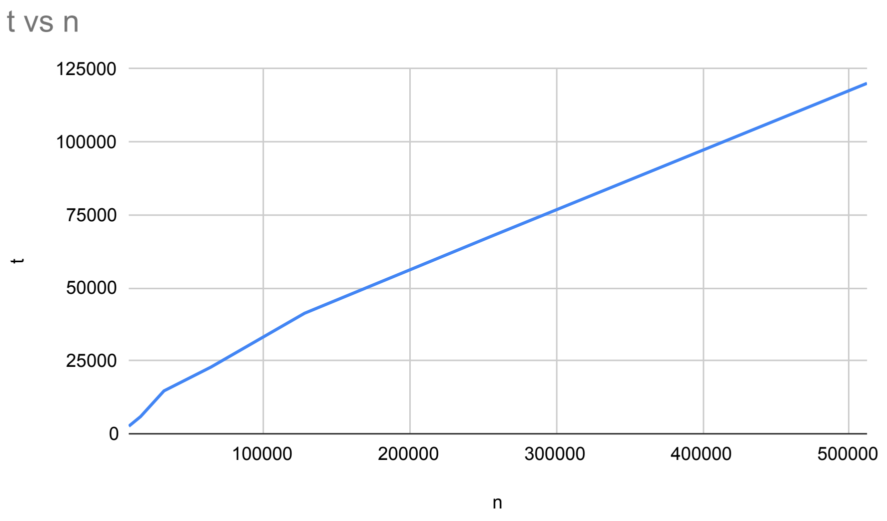

Figure 1: Graph for the search unsorted algorithm.

Search Sorted

In this algorithm, we make a slight optimization to the previous algorithm by adding an if clause that checks if the key even exists in the array. We do this by checking if the current array element is greater than the key value. If this condition is met this means that since the array is pre-sorted there is no way that the key element we are searching for exists in this array by any chance. This means we should return a false value to indicate that the array does not contain the key value.

Case	n	Time(in microseconds)	Ratio of time with the case 1 value as a reference
1	8000	170	1
2	16000	378	2
3	32000	770	5
4	64000	1417	8
5	128000	2760	16
6	256000	5582	33
7	512000	12060	71

This result was obtained upon doing the same algorithm for 100 iterations.

Binary Search

This algorithm is a classic example of a "divide and conquers" algorithm. Here we use the example of searching for a word in a dictionary as a practical example

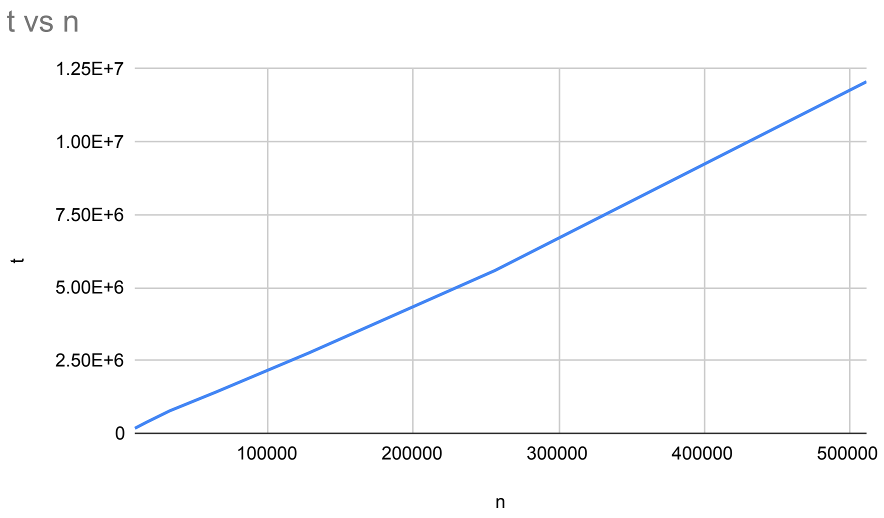

Figure 2: Graph for the search sorted algorithm.

of how this might be implemented. We start searching from somewhere close to the middle of the data set and then depending on if the middle value is greater than the key or vice versa. This way we keep changing the value for the mid-point for the search depending on our current position and search value. We do this while the value of the subtraction of the min and the max value is at least greater than 1. Aside from this, we check if the middle element happens to be the key element we are looking for and then we should return true to indicate our find. If the element at the middle position happens to be less than the key while the value of the middle position is less than the value for the max variable. Similarly, if the element at the middle position is larger than the key value and the middle position is greater than the value for the min variable. We had been provided with a basic skeleton for this algorithm which we then completed to complete the binary search algorithm. This has been implemented in the following piece of code below:

```
public static boolean binary_search(int[] array, int key) {
    int min = 0;
    int max = array.length-1;
    while (max - min > 1) {
    // jump to the middle
        int mid = min + (max - min)/2;
        // gives the middle point in the array's index
        if (array[mid] == key) {
            //System.out.println("key found");
            return true;
        }
        if (array[mid] < key && mid < max) {
        // The middle position holds something that is less than
        // what were looking for, what is the min possible page?
            min = mid + 1;
            continue;
        }
        if (array[mid] > key && mid > min) {
        // The mid position holds something that is larger than
        // what were looking for, what is the max possible page?
            max = mid;
            continue;
        }
        }
    return false;
}
```

The benchmarking data for the above algorithm has been tabulated below:

Case	n	Time(in microseconds)	Ratio of time with the case 1 value as a reference
1	8000	0.348	1.00
2	16000	0.568	1.63
3	32000	0.792	2.27
4	64000	1.010	2.90
5	128000	1.335	3.83
6	256000	1.879	5.39
7	512000	2.453	7.04

From the above data, we can deduce that this algorithm has a runtime complexity of the form `O(n) = a(log(n)) + b`. From this and the data in the table right above us, we can tell that the time taken for this is a lot lower than the previous algorithms we looked i.e. linear search. This result was obtained upon doing the same algorithm for 10000 iterations. Thus from this, we can conclude that the binary search is significantly more efficient than a normal sequential search algorithm (search unsorted) or even its optimized counterpart (search sorted). Similar to how we predicted the runtime for 1 million elements for the search unsorted algorithm, we can do the same thing for the binary search algorithm. We see that the general trend in this algorithm is that the runtime increase by a factor of 1.2 for an increase in the data size by a factor of 2. This means that time taken for 1 million elements should be close to 2.943 microseconds.

`Time taken for 512000 elements * 1.2 = Time Taken for 1 million elements = 2.943 microseconds(approximate)`

In the same way, we can also estimate the runtime for a dataset of 64 million elements. `Time taken for 512000 elements * (1.2) * (1.2) * (1.2) * (1.2) * (1.2) * (1.2) * (1.2) * (1.2) = Time Taken for 64 million elements = 8.7895425024 microseconds (approximate)`

Here we multiply the average ratio of 1.2 seven times to the 512000 n case, as 64 million would be roughly seven cases after the 512000 case. This is the reason for how we get the estimate of the time taken for 64 million values.

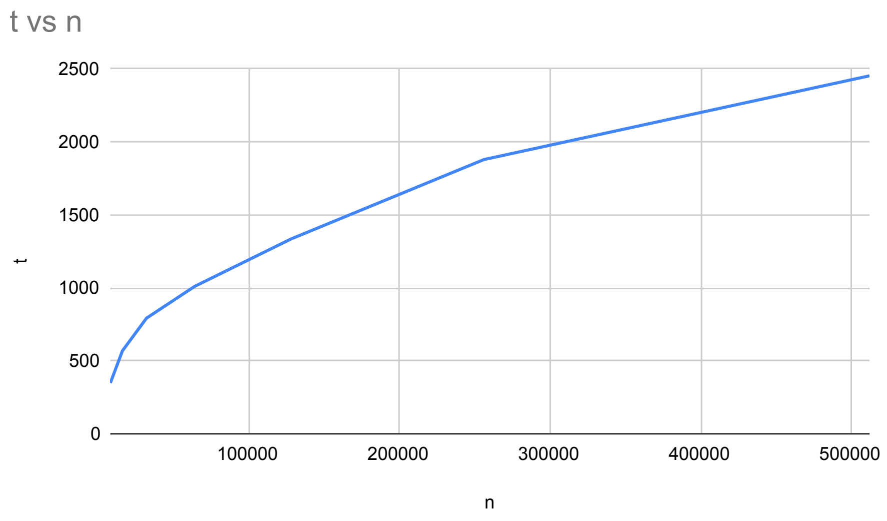

Figure 3: Graph for the binary search algorithm.

Binary Duplicates

Here we try to find duplicates using two arrays of the same length using one of the arrays as key values and the other array as the data set in which you search for the keys. We do this in the same way as we did for last week's assignment where we had to find duplicates using a sequential search to find the keys, but we replace the sequential search with a binary search. We use two nested for loops where the outer loop is to iterate through all the possible key values and the inner loop is used to iterate over the binary search part for (number of iterations = 100). The benchmarking data for the above algorithm has been tabulated below:

Case	n	Time(in microseconds)	Ratio of time with the case 1 value as a reference
1	8000	443	1.00
2	16000	950	2.14
3	32000	2242	5.06
4	64000	4992	11.26
5	128000	10548	23.81
6	256000	21846	49.31
7	512000	45345	102.35

We see a clear linear trend in the time taken to search for the keys. Compared to the time taken to search for the duplicates using sequential search which ended up having a quadratic time complexity. Thus using binary search for the purpose of searching for duplicates is clearly quite a bit more efficient than the previous approach. From this, we can also see that having sorted data is better, as this means that we can use more efficient algorithms such as the binary search algorithm.

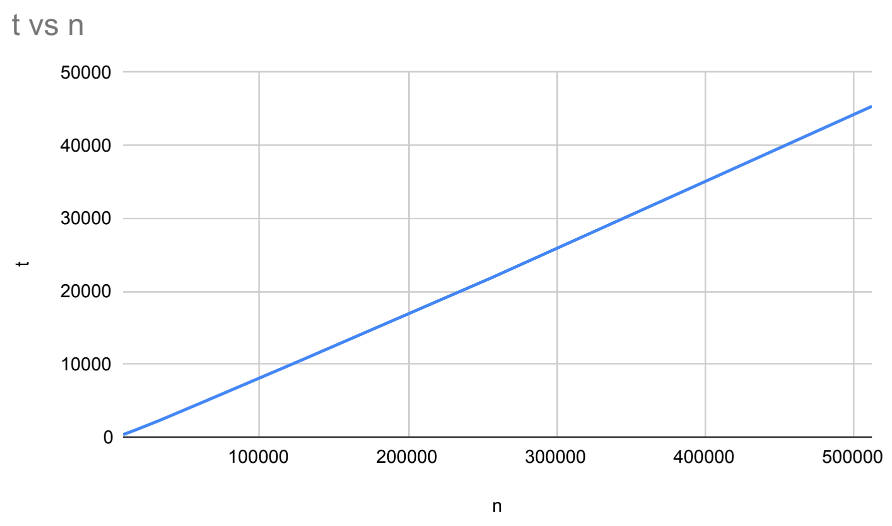

Figure 4: Graph for the binary duplicate search .

Parallel Duplicates

This algorithm tries to make an even more optimized way of searching for duplicates. We have two pointer variables that are used to keep track of the next values in both lists or arrays. This way we check if the next value in either one of the arrays is smaller than the other and depending on which value is smaller, we increment the list with the smaller element's pointer to the next index and so on. We do this while the value of both the pointers is less than the value of the last element's index. The benchmarking data for the above algorithm has been tabulated below:

Case	n	Time(in nanoseconds)	Ratio of time with the case 1 value as a reference
1	8000	54	1.00
2	16000	145	2.68
3	32000	351	6.50
4	64000	650	12.03
5	128000	1500	27.77
6	256000	2889	53.50
7	512000	5028	93.11

We obtained the following data upon iterating over this algorithm 10000 times. We see that the time taken by this algorithm also follows a linear trend. Thus the time complexity of this algorithm is given by O(n). We also notice that the time taken for this algorithm is significantly lower than using a binary search for the same task.

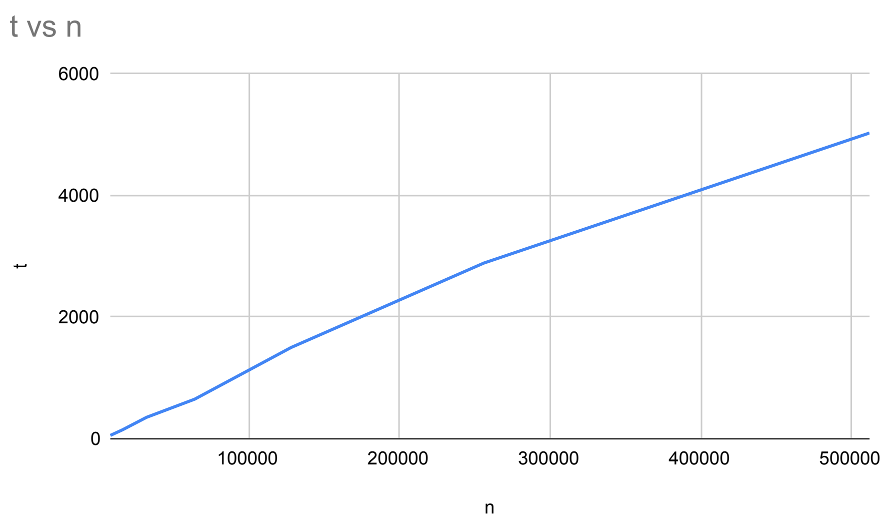

Figure 5: Graph for the parallel duplicate search .