Advantage of Sorted Data

Ayda Ghalkhanbaz

Fall 2023

Introduction

This lab mostly is about to analyse the time complexity of different searching algorithms in sorted vs unsorted arrays. The method in order to measure the time used in this lab generally is to search for a key in unsorted vs sorted arrays using linear, binary search and two pointers algorithms. Finally the results have been discussed.

Linear Search: Unsorted vs Sorted

The main purpose of the first task was to compare the time it takes to search for a key in both an unsorted and a sorted array of random values using a linear search.

Method

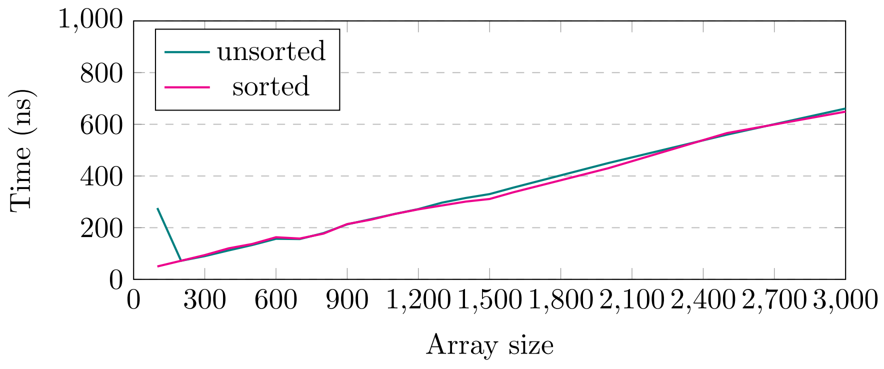
The implementation of the code of this task was straightforward since most of the code was provided through the assignment instruction, including the linear search method and the benchmark. The task was then completed by implementing a method for generating an unsorted array of random values.

Result

As demonstrated in figure 1, we can see the increase in the execution time as the length of arrays grows. The data plotted in this graph represents the average execution time for an array with size 'n' over $10,000$ searches with a new random search key for each search.

Also, the program was benchmarked for an array of size $1,000,000$. Executing a linear search on an unsorted array with one million random values took $218,368ns = 218\mu s$ while it took $207,197ns = 207\mu s$ for searching through a sorted array with the same size.

Figure 1. Linear search of unsorted vs sorted data



Discussion

According to the results above, we can observe that a linear search acts quite the same no matter if the data is sorted or unsorted. The logic behind this searching method is to search for the key in the array one by one which means that for an array of size ‘n’ it takes ‘n’ comparisons (in the worst-case scenario) to check if the key is found or not. In other words, the time complexity of linear search is $\mathcal{O}(n)$.

From figure 1, it is also clear that the sorted array mostly took shorter time than the unsorted array. The reason is their differences in the implementation of them. The advantage of having sorted data is that the program will check if the next element in the array is larger than the key and if it is the case, then it will stop searching. Thus searching linearly through sorted data can be potentially faster in practice even though time complexity remains the same, i.e $\mathcal{O}(n)$.

Binary search

In this task, a faster search algorithm than the linear search was implemented: the binary search. Binary search is a searching method where the key is sought within only half of the array and includes if-statements for comparison. It is remarkable to note that binary search is only applicable on sorted data.

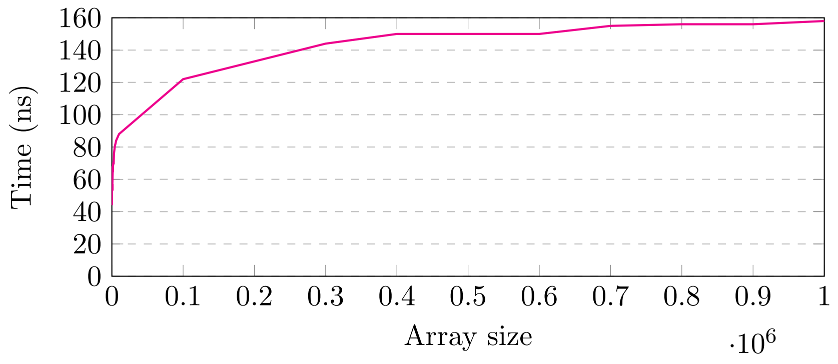
Method

In the same way as the previous task, the majority of the binary search method was provided in the assignment, with only a few additional details completed. For the benchmarking, a method similar to the one used to benchmark linear search of sorted data was implemented. The sorted array and the keys to be sought were generated by random values as well.

Result

The results of benchmarking the binary search for 10000 runs are shown in figure 2. As we can see the execution time has a logarithmic growth pattern. By utilizing Excel, the equation $t(n) = 13.461 \cdot ln(n)$ was obtained which represents the behaviour of the algorithm.

Figure 2. Binary search for sorted data



Discussion

As shown above in figure 2, it can be claimed that the time complexity of binary search is $\mathcal{O}(log(n))$. From the function presented in the result section, we can get an approximation of the execution time for binary search of an array with size 1000000, which is $13.461 \cdot ln(10^6) = 143ns$. This is quite an accurate approximation since it took about $158ns$ in practice. Using the formula again, we can get an approximation of time with 64M large data that is $13.461 \cdot ln(64 \cdot 10^6) = 1477ns$. Running a benchmark with this size of data gives $296ns$. In this case the approximation was not accurate since many factors affect the result we get in reality; such as cache, CPU, JVM optimization, algorithm behaviour etc.

Duplicates: Linear, Binary or Two Pointers?

The final part of this assignment is dedicated to implementing three different algorithms in order to find duplicates in two arrays. In this task we analyse these algorithms and discuss their time complexity and whether it is worthy to sort arrays before searching through them.

Method

For this task three methods were implemented. The first one is a linear algorithm which utilizes a nested for loop. The outer loop iterates through the first array and the inner loop iterates through the second array and for each duplicate encountered, a counter increments by one.

The second algorithm utilizes the binary search method implemented in the previous task. This method employs a for loop iterating linearly through the first array and uses its elements as parameters for the binary searches in the second array. For each duplicate encountered, a counter increments by one.

The last but not least algorithm is called the Two Pointers method. The advantage of this method is instead of having loops through arrays and searching for duplicates linearly/binary, elements in these sorted arrays will be compared simultaneously. The code snippet below shows the implementation of this algorithm.

```
while(ptr1 < array1.length && ptr2 < array2.length){
    if(array2[ptr2] < array1[ptr1]){
        ptr2++;
        continue;
    }
    else if(array2[ptr2] == array1[ptr1]){
        countDup++;
        ptr1++;
        ptr2++;
        continue;
    }
    else if(array2[ptr2] > array1[ptr1]){
        ptr1++;
    }
}
```

Result

The results of benchmarking the three methods mentioned earlier displayed in figure 3. The data plotted in this graph represents the average execution time of finding all duplicates in two unsorted arrays using the first method (linear) and two sorted arrays using the binary and two pointers method.Figure 4 provides a closer view of the two pointers graph since its behaviour was not easy to see in figure 3 due to scaling.

Figure 3. Duplicate search algorithms

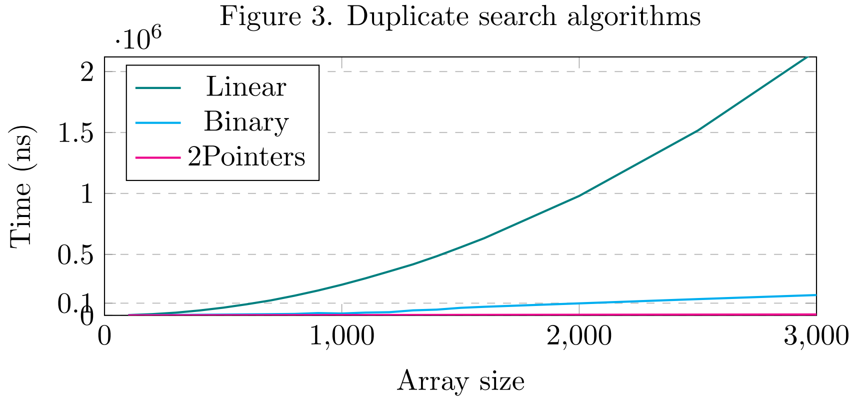
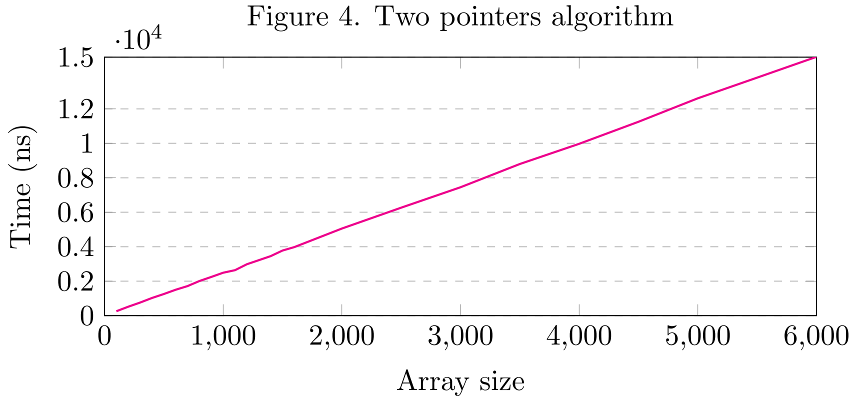


Figure 4. Two pointers algorithm



Discussion

The results obtained from figure 3 and 4, verify the theoretical time complexity of the methods in this task. As expected, the time complexity for the first method is $\mathcal{O}(n^2)$ since it includes a nested for loop and seeks for duplicates linearly. The time complexity for the second method is $\mathcal{O}(n * log(n))$ as it loops over the first array and searches for the duplicate using binary search. Finally , the time complexity of the two pointers algorithm is $\mathcal{O}(n)$ since it searches for similar elements simultaneously resulting in a linear in time complexity.

In conclusion, we can observe that the two pointers method is significantly faster than the first method. Thus, sorting data beforehand is not only efficient but also saves time and energy.