Sorted Data Assignment

Serban Alin-Cristian

September 8, 2023

1 Introduction

In this assignment I coded a linear and binary search algorithm and ran them through the benchmark provided by the assignment. Also, well, I had to modify the benchmark code to work in C, and deal with some difficulties regarding the binary search algorithm. But there really is not much to say here.

2 Efficiency of sorted data, in general

2.1 Unsorted data searching

Unsurprisingly, trying to look for a number in an unsorted array takes a very long time. Especially longer if the number isn't actually there, and you have to go through the entire array for it. Stopping when you find the number you need is a little bit better, but by a random amount each time you run it. This is not good at all and should only be done on very small datasets.

2.2 Sorted data searching

Now, when we have a sorted array, things become much easier. Our old method of going through each element and stopping when we find our number can become a little better if we also stop when we go over where the element should be. This is quite an improvement, but not as much of an improvement as using binary search. Linear search has an O(n) on average, but binary search has a O(log n)! Which is, in comparison, much faster. It works by basically splitting the array into halves and discarding the half that is too big/small for the key to be in, until eventually there is no more space it could possibly be in, or we find our key. Of course, everything said is under the assumption of already having the sorted data. Starting with unsorted data and then sorting it is a whole other domain that, while not covered by this report, still is very interesting.

3 Implementation

3.1 Linear

The linear code is not particularly interesting. Here it is for brevity.

```
{
    for (int i = 0; i < array.Length ; i++)
    {
        if (array[i] == key)
            return true;
    }
    return false;
}
```

3.2 Binary

In the scope of making this report look nice, here is the code:

```
int first = 0;
int last = array.Length-1;
while (true)
{
    int index = (last+first)/2;
    if (key == array[index])
        return true;
    if (key > array[index])
        first = index+1;
    else if (key < array[index])
        last = index-1;
    if (first > last)
        return false;
}
throw new Exception("uhhhhhh array exited out of infinity ig?");
```

This is the nicest implementation I could make, but I encountered a lot of difficulties trying to use the template from the sorted.pdf. Firstly, the requirement for index to be higher or lower than the first and last possible element can lead to "softlocking" the algorithm and running forever inside the while loop when you had an element that wasn't in the array. Secondly, it has no clause except if you actually find the key, so if the key is not in there then it softlocks again. This, however, almost feels intended as to make us students think for ourselves, but still led to a few minutes of confusion as to why my code ran forever.

3.3 Benchmarks

Adapting the benchmark program into C did not take much time at all (java and c are very similar anyway) but whatever the 789 gibberish was did not parse properly so it's gone. Also, c does not have nanotime, so instead I used the stopwatch elapsedtime, which is about as accurate anyway. I used a loop value of 10000 and k of 1000, and measured the total time for all the searches to complete for each size from 100 to 1600, jumping every 100. Thus, I present the following table for linear and binary searches.

Array Size	Linear Time	Binary Time
100	1.9530s	0.8204s
200	3.7199s	0.9092s
300	5.5090s	1.0130s
400	7.3260s	1.0315s
500	9.1094s	1.0483s
600	11.5714s	1.1924s
700	13.0088s	1.1224s
800	14.8341s	1.1246s
900	16.3721s	1.1703s
1000	18.8738s	1.2700s
1100	20.3844s	1.2726s
1200	22.3733s	1.2699s
1300	25.2044s	1.2319s
1400	25.8006s	1.2308s
1500	27.2231s	1.2301s
1600	29.5458s	1.3361s

Table 1: A table of total search speeds.

As is clearly visible, linear time slows down much more than binary time as size increases. This is because Linear is O(n), and Binary is O(log n). The reason this happens is because in binary search

we perform a lot less operations than linear, especially with large datasets.

4 Conclusions and takeaways

In the end, I've coded, successfully benchmarked and analyzed the results of both searches. There is not much else to say.