Searching in a sorted array

Axel Månson Lokrantz

Spring Fall 2022

Introduction

In this assignment the students were asked to work with sorted and unsorted arrays. Searching for a key in an unsorted array can be extremely time consuming, to see just how time consuming the students were asked to benchmark the arrays using different types of search algorithms.

The program was written in Java in collaboration with Pontus Kinnmark.

Task 1: Unsorted array

To start things off we began by benchmarking an unsorted array. To avoid running into issues with the cache the unsorted array and the key was randomly generated each loop iteration. For this benchmark we used a simple search algorithm that went through the whole array until the key was found.

```
for(int i = 0; i < loop; i++){
    for (int j = 0; j < unsortedArray.length; j++)
            unsortedArray[j] = rnd.nextInt(n/10);
    int key = rnd.nextInt(n/10);

    long t0 = System.nanoTime();
    search(unsortedArray, key);
    t_access += System.nanoTime() - t0;
}
```

n	ns	ratio
100	39	1
200	42	1.1
400	53	1.4
800	70	1.8
1600	112	2.9
3200	185	4.7
6400	319	8.2
12800	595	15.3

Table 1: Benchmark for an unsorted array.

As seen in the table and when plotting the results as a graph we get an algorithm with the performance of $0(n)$.

Task 2: Sorted array

In task 2 we ran the same benchmark test but this time with a sorted array. Same as before the array and key was randomly generated. Since we were searching through an array that we knew was sorted we could make the algorithm faster by stop searching for the element if the key value was smaller than the compared value.

n	ns	ratio
100	35	1
200	36	1.02
400	36	1.02
800	49	1.4
1600	53	1.5
3200	75	1.7
6400	128	2.7
12800	282	8

Table 2: Benchmark for an unsorted array. Run time in nanoseconds.

For a small n the new algorithm performs about the same, but for a growing n it is obvious that searching through a sorted array is nearly twice as fast. When plotting the numbers in a graph we can see a linear increase in data / time meaning we have the same performance of $O(n)$.

Task 3: Binary Search

In the third task we were asked to work with an even faster algorithm namely binary search. In short what binary search does is that it starts looking in the middle of the array then jumps either one quarter forward or backwards. The jumps become smaller and smaller until the searched element is found. The basic structure of the algorithm was already written but with a few gaps where the students were to fill in the missing code.

```
public static boolean binarySearch(int[] array, int key) {

        int first = 0;
        int last = array.length-1;
        while (true) {

        int mid = first + (last - first)/2;

        if (array[mid] == key) {
            return true;
        }
        if (array[mid] < key && mid < last){
        first = mid + 1;
        continue;
        }
        if (array[mid] > key && mid > first){
        last = mid - 1;
        continue;
        }
        return false;
        }
    }
```

To see just how much faster binary search was we once again had to run our benchmark tests.

n	ns	ratio
100	51	1
200	58	1.1
400	66	1.3
800	77	1.5
1600	86	1.5
3200	103	1.7
6400	125	2.4
12800	159	3.1

Table 3: Benchmark for binary search algorithm.

For a smaller n binary search is about the same as before, but as n grows larger we see a huge improvement compared to previous benchmarks. When plotting the numbers in a graph we see that the performance is $O(log(n))$. The polynomial based of the results is: $y = 22ln(x) - 62,8$. Searching through an array of 1 million items would take approximately 241 ns. While searching through an array of 64 million items would take approximately 332 ns.

Task 4: An even better algorithm

In the fourth and final task we were asked to do two new benchmark tests for the duplicate algorithm from assignment 1. Originally we used a O(n^2) algorithm. In the first benchmark we used binary search with with a performance of O($log(n)$). As seen in table 4 binary search is almost four times faster for n = 1280. Since we are using O($log(n)$) on n number of elements, the new performance becomes O($n * log(n)$).

n	n^2	$n * log(n)$
80	2169	4450
160	8660	9330
320	27530	22389
640	104205	50061
1280	394474	101518

Table 4: Benchmark comparison. Runtime in nanoseconds.

In benchmark 2 we tried to push the performance even further with a tailor made algorithm which can be seen below.

```
long t0 = System.nanoTime();
for (int i = 0, j = 0; i < n && j < n;) {
    if (first[i] < second[j]) {
        i++;
        continue;
    } else if (first[i] > second[j]) {
        j++;
    }
    else if (first[i] == second[j]){
        i++;
        sum++;
    }
}
    t_total += System.nanoTime() - t0;
```

n	ns
80	121
160	338
320	485
640	474
1280	2675

Table 5: Benchmark for the improved algorithm.

The new timings are almost incomparable with our previous ones which proves how much of a difference a few lines of code can make, especially for large arrays. Plotting the numbers gives an algorithm of $O(n)$.