Advantage of sorted data

Emanuel Paraschiv

Fall 2024

Introduction

Sorting elements is a procedure that can greatly influence the execution time. We will take some cases into consideration: First we will try to search for a key in an unsorted array and observe its behaviour. Then we will sort the array and observe the upgrade, and in the end we will implement a performant algorithm, better than linear search to get the smallest execution time for a search.

Unsorted search

The first benchamark we will run is on an algorithm that searches through each element in a sequencial order, called `unsorted_search()` . We can guess that this performance is dependent on the array size. Thus the complexity can be described as O(n). In order to run the measurement I created an array with random elements with the method `generateArray()`:

```
int[] array = new int[n];
for (int i = 0; i < n; i++)
    array[i] = rnd.nextInt(n * 2);
```

Then, a `benchmark()` method uses this array in a call to the algorithm that does the search, and measures the execution time:

```
long startTime = System.nanoTime();
unsorted_search(array, key);
long endTime = System.nanoTime();
```

In the end, the method returns the elapsed time:

```
return endTime - startTime;
```

After running the benchmark, we can see the results attached below.

As we can see, for an array with 1M elements, the execution time is about 500 μs. We are keeping this number as reference for the next implementations.

Input Size	Time (µs)
1000	3
10000	4
50000	23
100000	43
500000	270
1000000	540

Table 1: Execution time in Microseconds

Sorted search

In order to obtain a shorter execution time, one would think of ordering the elements inside the array. This way, we know that if we pass by an element that is larger that the one we are looking for, then we can finish the execution directly. The complexity of this algorithm remains however dependent on the size, n.

In order to to the measurements, we first create a sorted array:

```
int nxt = 0;
for (int i = 0; i < n; i++)
    nxt += rnd.nextInt(10) + 1;
    array[i] = nxt;
```

This array is used by the `sorted_search()` algorithm described earlier, and as before the measurements are performed by a benchamrk method which returns the elapsed time in ns:

```
long startTime = System.nanoTime();
sorted_search(array, key);
long endTime = System.nanoTime();
```

Input Size	**Time (µs)**
1000	0.3
10000	2.5
50000	13
100000	31
500000	180
1000000	428

Table 2: Execution time in Microseconds

As we can see now, for an array with 1M entries, the execution time decreased to around $400\mu s$ from $500\mu s$ previously. But there is still room for improvement.

Binary search

One step even further is to enhance the way we are searching. Going through each and every element from the beginning in a sequential order can be ineffective. One better approach is to implement an algorithm that can 'jump' depending on the element that it has to find. One such implementation is the `Binary Search`:

This algorithm focuses on 3 pointers: one at the beginning, one in the middle (called index) and one at the end.

As the array is sorted, we compare the element at position index with the one we search for:

- If the element is equal, then we return true as we found it:

```
if (array[index] == key)
        return true;
```

- If smaller, we split the array and continue searching in the left half:

```
else if (array[index] > key && index > first)
        last = index - 1;
```

- If bigger, we split the array and continue searching in the right half:

```
if (array[index] < key && index < last)
        first = index + 1;
```

Input Size	Time (µs)
1000	0.3
10000	0.4
50000	0.5
100000	0.65
500000	1
1000000	1.2

Table 3: Execution time in Microseconds

Compared to the previous cases, the execution time diminishes a lot: for 1M entries in an array, the time is only around 1 μs.

We can also see that the execution time increases at a decreasing rate as we are halving the array every time we do a search.

Performance is $O(log(n))$ as execution time grows logarithmically with the array size.

I guessed that for an array of size 64M the execution time could be around 1.5 μs.

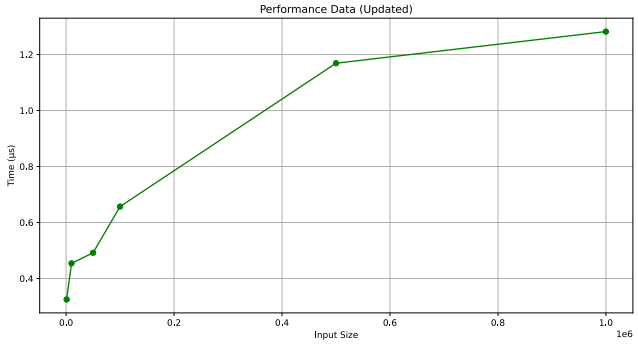

Figure 1: Execution time for Binary Search

However, the result proved to be 2.3 μs :)

With the help of an online calculator, I was able to come up with the following function to describe the execution time y given the array size x:

$$y = 14 \cdot \log(x) - 82 \tag{1}$$

Recursive programming

With all these changes so far, we can continue by making our algorithm recursive. This way the binary search algorithm will call itself each time we need to change the pointers, until we find the specific key.

The situation in which we don't perform the call are called the `Base Cases`:

- If the middle element is equal to the key, return true:

```
if (arr[index] == key)
        return true;
```

- If the range is invalid (i.e., $min \geq max$), return false.

Otherwise, he have the `Recursive Case`:

- If the middle element is greater than the key, we search the left half by adjusting the max index:

```
if (arr[index] > key)
  return recursive(arr, key, min, index - 1);
```

- If the middle element is smaller than the key, search the right half by adjusting the min index:

```
    else
      return recursive(arr, key, index + 1, max);
```

This binary search works in $O(log(n))$ time complexity because the search space is halved at every step.

Because each recursive call reduces the search space by half, the number of recursive calls is approximately how many times we can halve the array size until it reaches 1.

This is given by log2 (n), where n is the size of the array.

For example:

- Array size of 1000: $\log_2(1000) \approx 10$ recursive calls
- Array size of 2000: $\log_2(2000) \approx 11$ recursive calls
- Array size of 4000: $\log_2(4000) \approx 12$ recursive calls
- Array size of 1,000,000: $\log_2(1,000,000) \approx 20$ recursive calls

It follows thus that the number of recursive calls in binary search is not a problem. This is happening as the number of recursive calls grows logarithmically with the array size, meaning it increases very slowly as the array size grows. This is far from being an issue for the call stack, which can typically handle thousands of function calls without any problem.

However, recursive strategies can become problematic in other algorithms with large recursion depths. In those cases, recursion should be avoided or optimized with techniques like tail recursion or converted to iteration (like in the previous task).