Runtime and Time Complexity of Search Algorithms on n Sized Arrays

Malte Berg

HT 2024

Introduction

This report covers the third assignment assigned in the course ID1021. The assignment is to analyze the runtime and time complexity for different search algorithms on arrays of different sizes. The report will cover a linear search algorithm on sorted and unsorted arrays as well as binary and recursive search algorithms on sorted arrays only.

Linear Search Algorithm

Unsorted Array

For the first task, an n sized unsorted array was to be searched. A very simple search algorithm can be implemented without much logic to perform this search.

```
public static boolean unsorted_search(int[] arr, int key) {
    for (int k : arr) {
        if (k == key) return true;
    }
    return false;
}
```

The code above is a way of searching an array linearly. It starts by checking if the first element is equal to the wanted key. If not, it moves on to the next element. It continues like this until the key is either found, returning `true`, or until the end of the array is reached with no key found, returning `false`.

The array sizes chosen are ranging from 2^7 to 2^{23} with non-duplicate integers randomly created with the following code below.

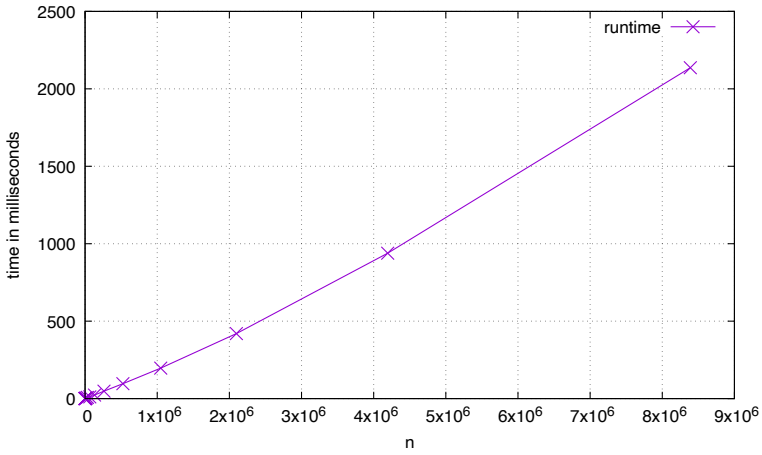

Figure 1: Linear search on unsorted array of size n

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

This code generates random integers in a sorted array of size n. To use an unsorted array, the Fisher-Yates shuffle is used on the array before using the search algorithm.

As can be seen in Figure 1, the runtime increases linearly as the array size increases. From this, it can be stated that the time complexity of the linear search algorithm is $O(n)$.

Sorted Array

The code from the previous section for creating a sorted array is now used without the shuffle algorithm. How does the runtime and time complexity differ by searching linearly on a sorted array? The hypothesis for this report is that the runtime will be almost identical as there is no further logic built in to the search algorithm benefiting off the fact that the array is now sorted.

Comparing Figure 1 and Figure 2, there is almost no difference at all in runtime, supporting the hypothesis. This is because there is no added logic to the search algorithm to take use of the fact that the array is sorted.

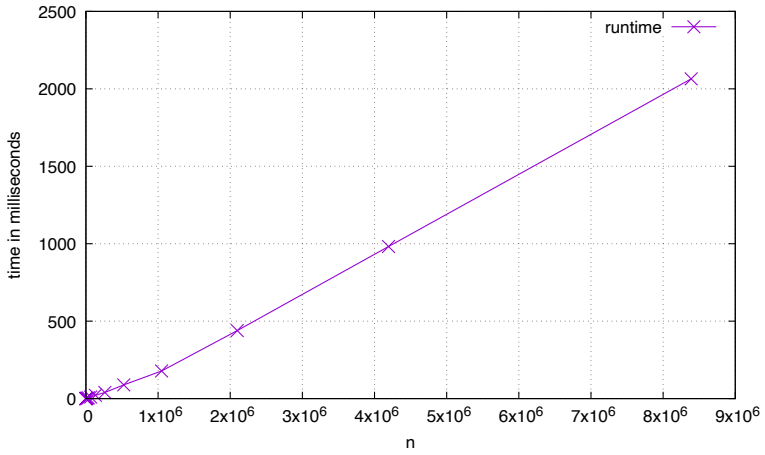

Figure 2: Linear search on sorted array of size n

One way to reduce the runtime would be to stop the search if the element compared to the key is larger than the key, and to then return false as the next element can't possibly be smaller and therefor will not equal the key. However, this will only slightly reduce the runtime and will not change the time complexity, which for the linear search algorithm on both the sorted and unsorted arrays will be just $O(n)$.

Binary Search

As the program now works with sorted arrays instead of unsorted arrays, it should use a search algorithm that benefits from searching on sorted arrays.

```
public static boolean binary_search(int[] arr, int key) {
    int first = 0, last = arr.length - 1;
    while (true) {
        int index = (first + last) / 2;
        if (arr[index] == key) return true;
        if (arr[index] < key && index < last) {
            first = index + 1;
            continue;
        }
        if (arr[index] > key && index > first) {
            last = index - 1;
            continue;
        }
        return false;
    }
}
```

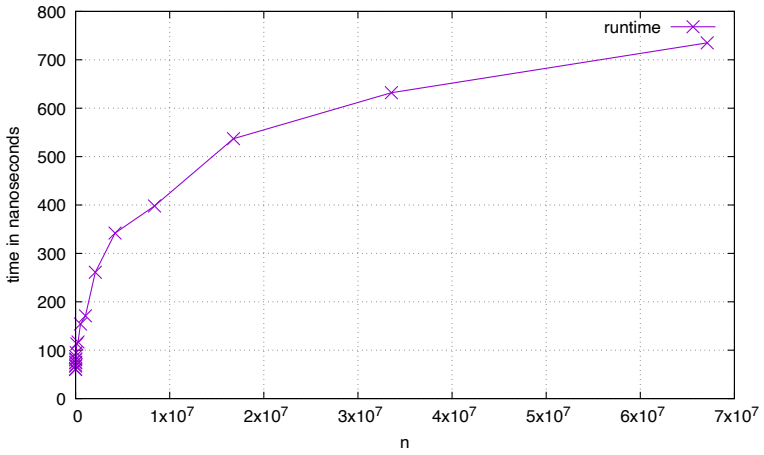

Figure 3: Binary search on sorted array of size n

One search algorithm where this is the case is binary search. The concept is that given an array and a key, a `first` and `last` index is initialized, and the middle is then looked at.

If the integer in the middle is the wanted key, instantly return true. If the integer is bigger than the wanted key, the `last` is changed to the middle index minus one, as if the key exists in the array, it is somewhere in the lower half of the possible range.

If the integer is smaller, `first` is instead changed to the middle index plus one, as the key should be found in the upper range. This is repeated until the key is found or until only two elements remain in the range and neither is the key.

The assignment asked how long the runtime was for an array size of 1 million as well as estimating the runtime for an array size of 64 million. The runtime for an array size of 2^{20}, which is approximately 1 million, came out to about 171 ns per search. A complete shot in the dark estimation for the 64 million array size was about 3000 ns, about $\frac{1}{4}$ times the increase of 64x. The true value was about 735 ns, beating the estimate by a lot.

As can be seen in Figure 3, the runtime increases quickly with the increase of the array size n in the beginning. However, as the size n of the array continues to grow, a trend can be seen where the runtime appears to level out. This graph clearly shows the nature of a logarithmic graph. For smaller sized arrays, the two algorithms are comparable, but for larger sized arrays the binary search algorithm is way more optimized. Notice also that the runtime for the linear search in Figure 1 and Figure 2 is in milliseconds while the Binary search instead is in nanoseconds. All in all, the conclusion for the binary search is that its time complexity is $O(log(n))$.

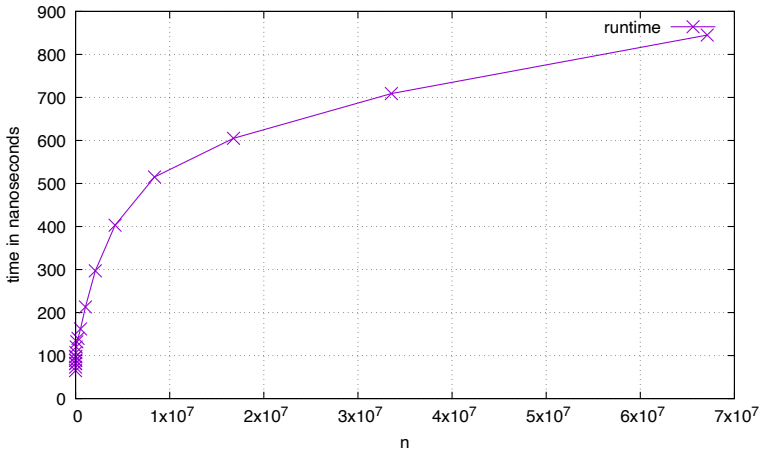

Figure 4: Recursive binary search on sorted array of size n

Recursive Binary Search

Instead of doing regular binary search, the program can instead run a recursive method. Instead of a while loop, the method calls itself with updated values for `first` and `last`.

```
private static boolean recursive(int[] arr, int key, int min, int max) {
    int mid = min + ((max - min)/2);
    if (arr[mid] == key) return true;
    if ((arr[mid] > key) && (min < mid)) recursive(arr, key, min, mid-1);
    if ((arr[mid] < key) && (mid < max)) recursive(arr, key, mid+1, max);
    return false;
}
```

One thing that can be noticed is the slightly higher runtime for the recursive binary search seen in Figure 4 compared to the regular binary search in Figure 3, however it remains the same time complexity at $O(log(n))$. The amount of recursive calls were also tracked with a global variable. The average amount of recursive calls per search were almost precisely $\log_2 n$. One can guess that doing recursive calls, with the big arrays as arguments provides strain on the system which increases the runtime.