Time complexity in search algorithms in C

Sam Serbouti serbouti@kth.se

September 11, 2024

In this report, I will explore the time complexity of various search algorithms implemented in C, and compare their efficiency. Starting with linear search in an unsorted arrays, optimizing it to searching in sorted arrays, then using a binary search, and finally a recursive method. For better comparison between algorithms, I will, for each benchmark, perform warm-ups operations of `benchmark(10000000)` (to fully optimize the cache), build two arrays : one for `n` items (which corresponds to the size of the array, and will grow on a log scale) the other for `numberOfSearches` keys (where `numberOfSearches` is 1000 and corresponds to the number of times I will run the search algorithms in the array to find a key).

```
int *array= (int*)malloc(n*sizeof(int));
for (int i = 0; i<n;i++) array[i] = rand()%(n*2);

int *keys = (int*)malloc(numberOfSearches*sizeof(int));
for (int i = 0; i<numberOfSearches;i++) keys[i] = rand()%(n*2);
```

To compare time complexities, I measure the total searchtime for these 1000 searches. But I will do so 10 times, and extract a minimum, maximum and average values of these 10 experiments. They loosely correspond to best, worst and average cases.

1 Search time in unsorted array

I start by a linear search through an unsorted array (ie I go element after element, one by one, in a series of items and compare them to the key I am looking for). For 1000 searches, the total runtime follows a power-law model (see Figure 1) which I can say is of ordo n:

$$t_{total,unsorted}(n) = 4 \cdot n^{1.005} \hookrightarrow \mathcal{O}((n))$$

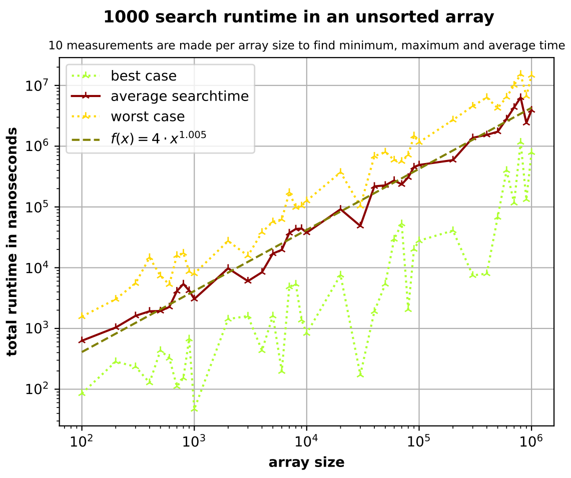

Figure 1: The total runtime follows a power law model

2 Sorted search runtime

I now sort the array so that the items are increasing (randomly) in value
`nxt += rand()%10 + 1;`

When going through the array, searching for the key, I can add an exit condition that allows to cut down the computing time:
`if (array[index] > key)return 0;`

I realise that it **doesn't change much** compared to when we had an unsorted array. The total runtime also loosely follows:

$$t_{total,sorted}(n) = 4 * n^{1.005} \hookrightarrow \mathcal{O}((n))$$

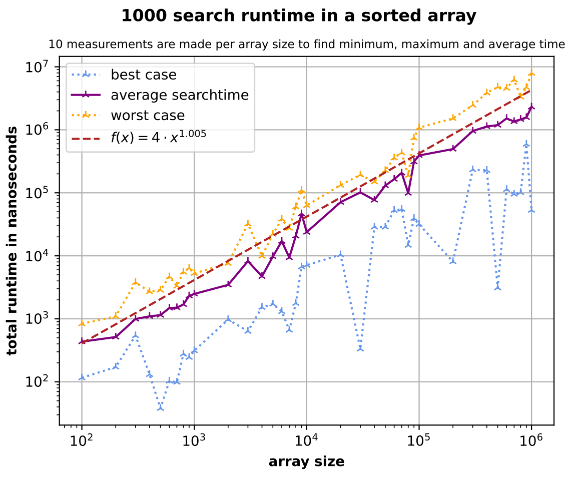

Figure 2: We don't benefit much from searching in a sorted array

	unsorted array	sorted array
total runtime	5392776	1420212
ratio	1.0	0.26

Table 1: Search times for a one million items array

3 Binary search runtime

A binary search starts at the middle element of the array, it compares it to the key and then narrows the search to the left or right half of the array, depending on whether the target is smaller or larger, until the target is found or the interval is empty.

```
int first = 0;
int last = length - 1;
while (1) {
    int index = first + (last - first) / 2; //jump to middle
    if (array[index] == key) {return 1;} //key found
    if (array[index] < key && index < last) {
        first = index + 1; //first pointer searches right half
        continue;
    }
    if (array[index] > key && index > first) {
        last = index - 1; //last pointer searches left half
        continue;
    }
    return 0; //empty interval
}
```

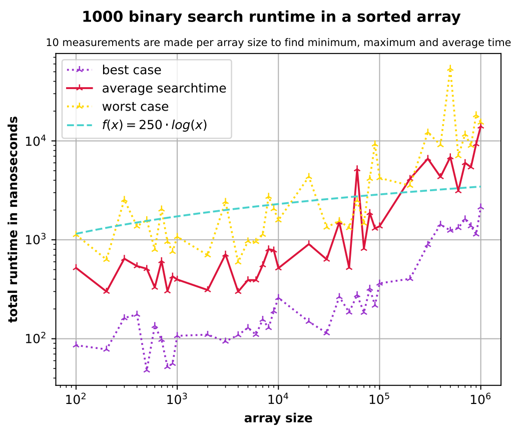

Figure 3: The binary search runtime follows $\mathcal{O}(log(n))$

I find an average time complexity of $t_{total,binary}(n) = 250 \cdot log(n)$ that follows $\mathcal{O}(log(n))$ and I can assume a runtime of $1951ns$ for a binary search in a 64 million items array. My machine takes $5487ns$ which is in the same order of what was expected.

4 Recursive search runtime

In a recursive algorithm, I have the same objective of narrowing the search step by step but this time, the function calls itself. This means I have to get rid of the `while(true)` loop otherwise the machine will get stuck in a `Segmentation fault` error:

```
if (array[mid] < key) {
    return recursive_search(array, length, key, mid + 1, last, call_count);
}
if (array[mid] > key) {
    return recursive_search(array, length, key, first, mid - 1, call_count);
}
```

The call_ count is a pointer I added that tracks the number of time the function is called and corresponds to the orange line in the plot Figure 4
I realise that it **doesn't change much** compared to the direct binary search. The total runtime follows:

$$t_{total,recursive}(n) = 0.01 \cdot n \cdot log(n) \hookrightarrow \mathcal{O}((nlog(n))$$

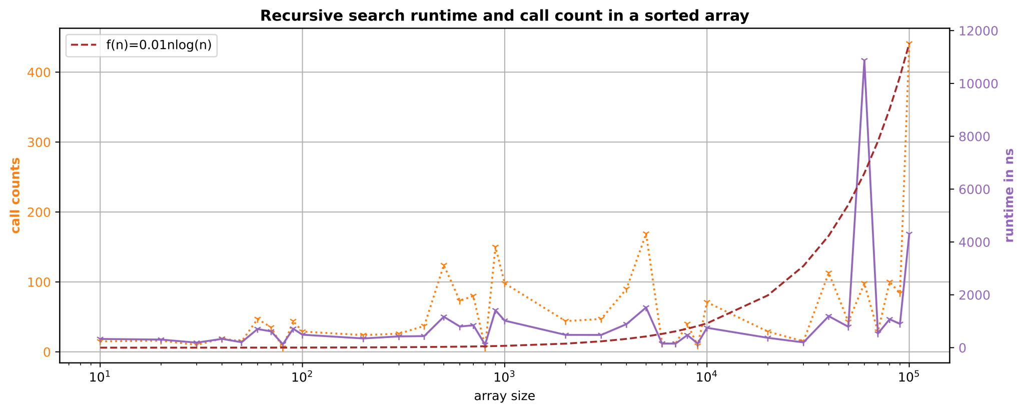

Figure 4: The recursive search time follows $\mathcal{O}((nlog(n))$