Arrays in C

Péter Herczku

Fall 2024

Introduction

The task is to analyze the performance and the time it takes to run array operations in C, such as randomly accessing an item, searching for an item and searching for duplicates.

Random access

It is crucial to know how to measure execution time accurately. We can use the built in clock in our computer for this purpose. In C, we can get the current clock time using the `clock_gettime` function. In the description of the assignment we were provided with the method `nano_seconds` that returns the elapsed time between two states in nanoseconds. We will use this function to evaluate execution time.

First we can measure the time it takes to run a single element access. After a few benchmarks, we can see that it varies between 0 and 300 nanoseconds. The reason this is a relatively large number is that we are actually measuring the time it takes to call `clock_gettime` rather than the array access itself. We can prove this assumption by only calling the `clock_gettime` function twice. After looking at the benchmarks, we get the same result: the execution time is 0–300 ns.

Instead, we could perform the operation n times, measure the time it takes to do that, and then divide this value by n. Additionally, for the access to be random, we need to set up another array with random indexes. We should do it before benchmarking the loop, otherwise we would measure the time it takes to generate a random number as well.

```
// ... initialize variables, fill up arrays
clock_gettime(CLOCK_MONOTONIC, &t_start);
for (int i = 0; i < loop; i++) sum += array[idx[i]];
clock_gettime(CLOCK_MONOTONIC, &t_stop);
```

attempt	runtime
1	2.0
2	2.1
3	2.0
4	2.4
5	3.2

Table 1: Results of the first benchmark, runtime in nanoseconds

Maximum, average and minimum time

We can see that the runtime varies significantly. In order to get a better understanding of how long it actually takes to run this operation, we can run it multiple times and measure the minimum, average and maximum time on different sizes of an array.

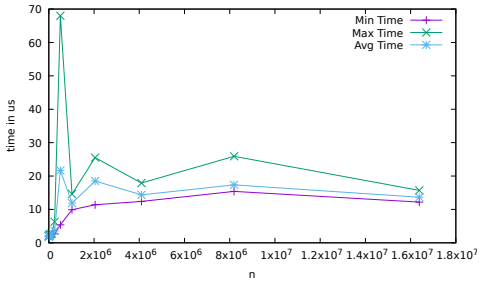

Figure 1: Graph of the minimum, maximum and average execution time on different sizes of arrays

We observe a spike in the beginning, then the graph seems to consolidate but still continues to increase. Let's take a closer look at the very beginning of the graph.

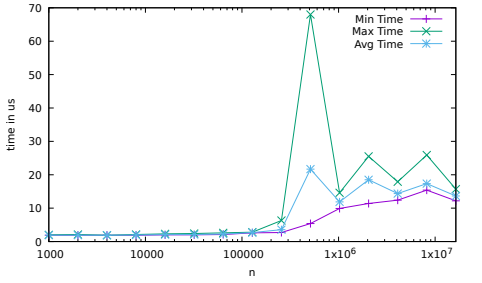

Figure 2: Graph of the minimum, maximum and average execution time on different sizes of arrays

One can notice that the average execution time is highly affected by the

maximum execution time. However, the maximum execution time can be arbitrarily large, since we are running the program on top of an operating system that could prioritize other tasks over ours, therefore we can neglect the maximum and average values. The minimum time seems to be consistent in the beginning, but then starts increasing. This might be an effect of several factors e.g cache misses.

Measuring median

To find out more information, we can measure the median since it would give us a clearer picture.

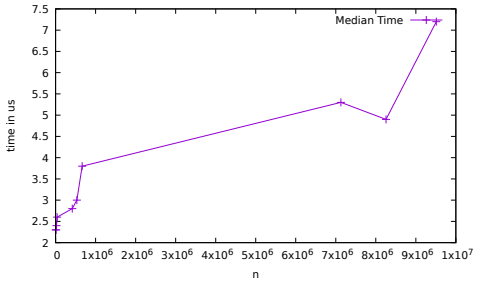

Figure 3: Graph of the median execution time on different sizes of arrays

The conclusion of this graph is that the size of an array actually have an impact on the execution time of a random access in C. If we are working with relatively small sizes of an array this difference is negligible, but not on large sizes. On the other hand, we can also see that the median and minimum values are pretty close to each other, therefore we can use the minimum time in future benchmarks.

Search for an item

In this task, we need to set up two arrays: one target array with random values and another one with keys that we want to search for. When we set up the benchmark, we need to take into consideration that the arrays should not be sorted, and the size of the `keys` array should be larger than the size of the target array. Otherwise we might get false results because if the size of the keys is very small it may result in quickly finding the element we are looking for. Let's run this benchmark with the provided method on different sizes of arrays, and analyze the results.

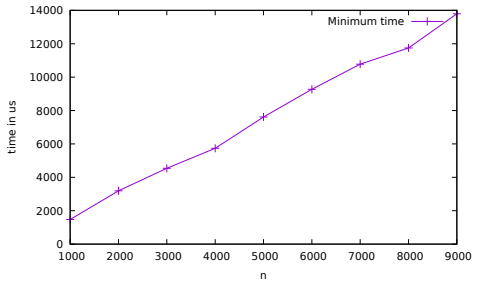

Figure 4: Graph of the minimum execution time for search

The graph clearly shows that as we increase the number of elements in an array the execution time grows linearly as well. In other words, the time complexity of this function is $O(n)$.

Search for duplicates

In this task, we have 2 arrays with the size `n`, and we need to find all elements that are contained by both of the arrays. With the given code from the assignment, we set up the benchmark accordingly. In theory, we iterate through 2 arrays n times, so the complexity should be $O(n^2)$.

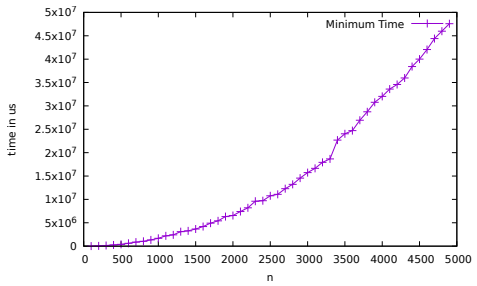

Figure 5: Graph of the minimum execution time for duplicates

Our graph indicates that our theoretical assumption was correct, and we got a quadratic relationship. The time complexity of this function is truly $O(n^2)$, meaning that if the method takes t time for an array with n elements, it takes $4t$ time for an array with $2n$ elements.

GitHub

I have uploaded the full project to my github repository, where we can find the code I used to make this report.

Conclusion

In conclusion, this was a fun report to write, I have learnt how to set up pipelines for LaTeX projects, and how to use LaTeX in general. Seeing time complexities not only theoretically has also helped me to understand it better.