Assignement 1 Report - Arrays

Sam Serbouti serbouti@kth.se 20010513-3284
TCOMK Algorithms and Data Structures ID1021 HT24

September 1, 2024

Clock resolution

Before finding benchmarks for several algorithms, I need to test my clock's resolution. I firstly measure the time it takes to turn on and off my clock 1000 times to get an average and compare it to the measurement of a simple fetch task `sum += given[i];`

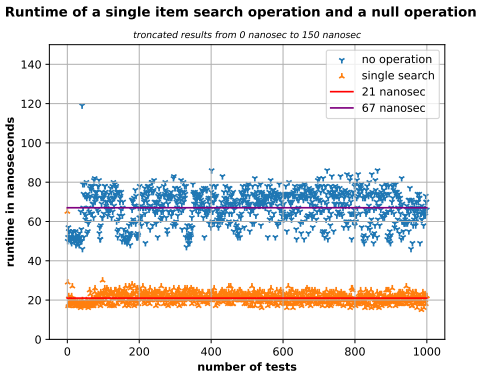

Figure 1: Comparing runtime of single operation and null operation

	min	*ratio*	mean	*ratio*	max	*ratio*
0 operation	46	*1.0*	67	*1.0*	212	*1.0*
1 operation	15	*0.32*	21	*0.31*	35	*0.30*

Table 1: Runtime of a null operation in nanoseconds in relation to the null operation

In conclusion, my clock's resolution (estimated at 67 nanoseconds) is **three times too small** to measure a single operation. I will need to do several operations and divide the runtime to get an accurate runtime measure.

Random array access

Multiple random array search

Now that we know we cannot do just a single operation and measure it. I will try to measure the runtime for $nbOfSearches = 1000$ and divide this runtime by 1000 but I will do so for 1000 tests and obtain: minimum value = 13 ns ; **mean value = 14 ns** ; maximum value = 25 ns.

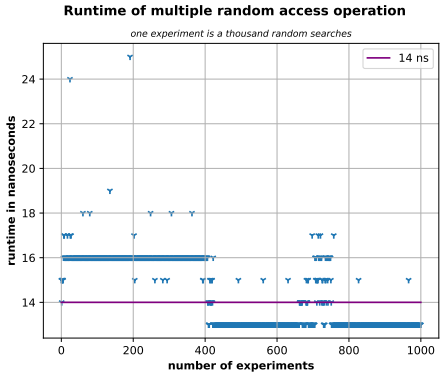

Figure 2: Random array access runtime per operation for 1000 searches

Changing array size

To evaluate the impact of an array's size in the runtime, we need to allow dynamic memory allocation, grow my array's size and measure min, max and mean values of runtime for 10 searches.

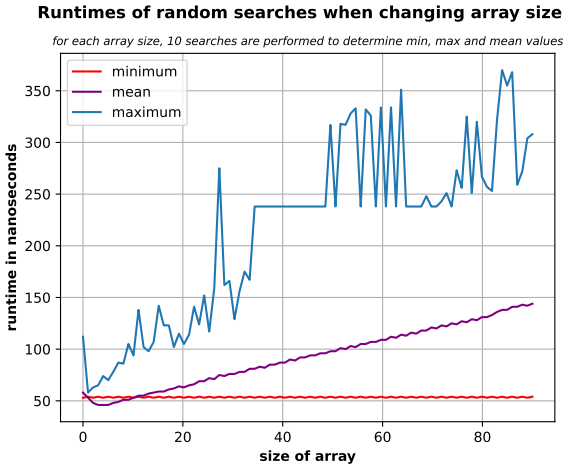

Figure 3: Changing the array size does not change minimum runtime

I firstly notice that the maximum value for runtime is volatile (due to the CPU managing other tasks) thus it is more pertinent to focus on minimum or median values. Moreover, changing the size of the array **does not impact the minimum runtime** for random array access for arrays with 100 or less items. It might not hold true for bigger arrays because the data can no longer be easily found in the cache.

Search for an item

I firstly wish to establish the minimum number of searches I should perform, at a fixed array size of 100 items, to get a stable runtime. See figure 3 (left) to notice that I need at least 100 000 searches. After that, I measure the time it takes to find 100 000 items (per item) in array depending on the size of the array n.

```
for each array of size from 100 to 100 000:
    start clock
    for 100 000 times:
        go through the array
            if you find the item, stop
    end clock
    divide time by 100 000
```

I find that the runtime per search operation follows a power-law :
$t_{item}(n) = 1.38 \cdot n^{1.02}$

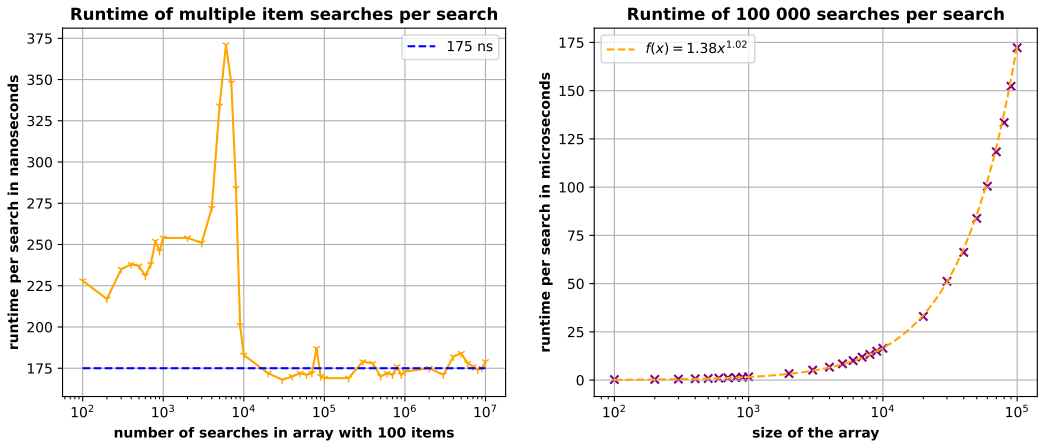

Figure 4: Item search runtime per search stabilizes after 100 000 searches (left) and the runtime follows a power-law model (right)

Search for duplicates

When searching for duplicates in two separate arrays of a growing size n, it is easy to get a buffer overflow `malloc(): corrupted top size Aborted (core dumped)` error when running. However when restricting to 100 searches (or loops within the array), I still manage to grow the array and find a power-law model : $t_{duplicate}(n) = 26.63 \cdot x^{0.77}$

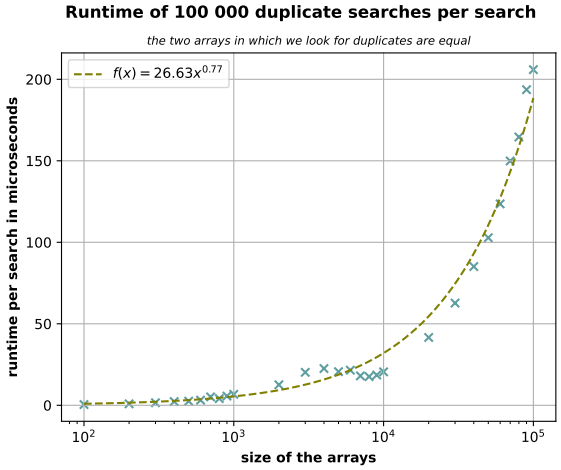

Figure 5: The runtime per duplicate search follows a power-law as well

Assuming my models remain accurate for bigger arrays, I can plot both functions (t_{item} and $t_{duplicate}$). Interestingly, before the array reaches 126000 items, finding a duplicate is more time-costly than finding a single item but not after that point.

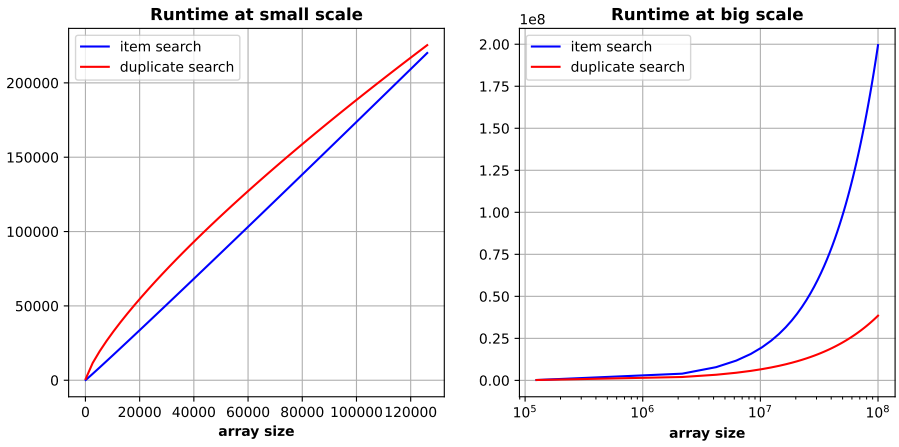

Figure 6: Plots of models at small and large scales