Assignment 1 : Introduction

Aditya Gupta

August 31, 2022

Introduction

This is my report on the bench-marking problem given in assignment 1 where we need to find and compare the run times of random access in an array, search for data in an array and finally try to find duplicate elements in two arrays. We solve this problem using the `System.nanoTime()`. We first try and check the accuracy of the java clock using which we will be conducting this experiment. We will then follow this up with how a method to randomly access or operate upon data from our target array and then find the time it takes and compare it for a different number of elements in the array. After a benchmark for checking the time taken has been chosen this would then be used to complete:

- Task 1: Test the time taken for random access and search for an item in the array.
- Task 2: Test the time taken for checking for duplicate elements in 2 arrays of length n.

Checking the Java Clock's accuracy

We check how accurate `System.nanoTime()` is in being able to tell how much time it takes to do a specific task. To test this we use this piece of code given inside the assignment document.

```
for (int i = 0; i < 10; i++) {
    long n0 = System.nanoTime();
    long n1 = System.nanoTime();
    System.out.println(" resolution " + (n1 - n0) + " nanoseconds");
}
```

We take a value of n from the user and then check how much time elapses between the two `System.nanoTime()` time checks before and after the summing of elements from the target array (which was done to avoid compiler optimizations). We compare this against the value of n and how it affects the time taken to do this task. Upon doing this we realize that we get erratic values for each

time the `System.nanoTime()` is checked. The process takes 100 ns, then 200 ns then, 400 ns, then 100 ns again and so on. This behaviour can be attributed to the fact that this experiment is being done on a computer with other background tasks taking place at the same time. These tasks are all queued one after the other before they get executed which might explain the time difference in doing the same task each time. This leads to the need for a better benchmarking process than just slapping on two `System.nanoTime()` time check calls.

Benchmarking

Now that we have seen the limitations of `System.nanoTime()`, we can start to make the benchmarking functions that we will use in Tasks 1 and 2 while keeping the previous observations in mind.

Task 1 : Random Access

To access elements randomly we initialize an array `index []` with random values between 0 and n. This array's elements are used as the indices for the target array so that when we access the elements of this array, the access is random rather than sequential which would have changed the time of execution because of caching. Keeping this in mind our benchmarking function gives us the time taken for random access by taking two separate time checks being :

1. t_{access} : t-access is the time taken to access elements randomly and add them to the variable sum.

2. t_{dummy} : t-dummy is the time taken to just do a normal add operation.

We check this for many different values of n and see how the time changes for each case as we did before. The observed values of $t_{access} - t_{dummy}$ for different values of n has been tabulated below:

Case	n	Time
1	1	0.95 ns
2	10	0.56 ns
3	20	0.266 ns
4	100	0.252 ns
5	1000	0.25 ns
6	10000	0.24 ns

Conclusion to benchmarking for the random access problem

We see that we get lower values for the overall time taken to do the process. This is something which confused me a lot as it didn't make sense as to why the overall time would decrease with the increase of the n value. My understanding of why this happens is that the compiler tries to make more optimizations with

the increase in the size of data. That seems to be the only reasonable way that the above result could make sense to me.

Task 2A : Searching for an item in a array

Here now that we have established a benchmarking system we can build upon it and try to find the time taken for an algorithm which searches for a key element in an array. We feed the benchmarking function two values, those being:

1. `k ((number of rounds))`
2. `m (number of search operations in each round)`

To be able to get predictable results I set the value of the number of rounds to 10 and I set the number of search operations to twice the value of n so that the result is within predictable bounds. This key value has been randomly generated and we choose one of the random key values for each iteration to search for. We search for the keys by comparing all the elements in the array to the key value to check if we have a match. If we find the key we break out of the loop immediately. We add the time taken in searching for all the keys and we return this value as the overall time. We try and find a relationship between the number of elements and time using the returned value. The returned values for this task have been tabulated below:

Case	n	Time	Ratio of the times with respect to the first time value
1	1	0.002 ms	1
2	10	0.012 ms	7
3	20	0.023 ms	15
4	100	0.354 ms	236
5	1000	13.02 ms	8700
6	10000	310.4 ms	200000

We observe that the values are increasing in a similar ratio to the one between the values of n for which the time is checked. For example: In cases 2 and 3, the ratio of time taken for searching the key is similar to the ratio between the values of n for the two cases. There are some inconsistencies to this as we can see in the table data.
A polynomial that can be used to define the time taken by the searching algorithm and also explain some of the inconsistencies in the time values can be represented using this function of n, given by the following equation: $t(n) = a(n)+b$
Here `a` and `b` are constants and `n` is a variable.

Conclusion: The ratio of Cases is similar to the ratio of their corresponding time taken and thus we can conclude that the search value

increases linearly with the increase in data values, which can be represented using the polynomial above.

Task 2B : Searching for duplicates

Now we use the concepts from the previous task and apply them here. Now we have the same number of randomized keys as the number of elements in the target array at all times. The time taken by the process to search for duplicates in the array is tabulated below.

Case	n	Time	Ratio of the times with respect to the first time value
1	1	0.001 ms	1
2	10	0.012 ms	9
3	20	0.033 ms	25
4	100	0.523 ms	402
5	1000	7.88 ms	6057
6	10000	442.4 ms	340000

We realize that the time taken in this case can also be represented with a polynomial i.e. the polynomial which we got in the previous case. This however is an inconsistency, as the Polynomial for representing the time taken for checking duplicates should be an order 2 polynomial of the type:

$$t(n) = a(n^2) + b(n) + c$$

where a, b and c are constants. This is because we are going through the array using 2 nested for loops to search for the duplicates which means that the number of operations used to do this is n times n or n^2. Thus this is an inconsistency.

Maximum value of n for 1 hour of runtime

We know that there are 3600 seconds in an hour, and each second has 1 billion ns, which means there 3.6 trillion ns in an hour. Now we know that if the time complexity of this algorithm is of the order of n^2 thus it means n^2 where the (n value corresponds to the number of operations) and which corresponds to the 3.6 trillion ns of time in an hour. If we assume that each operation takes 1 ns then $n^2 = 3,600,000,000,000ns$ which implies that $n = 1,897,366.6$. This means that the largest value of n for which the run time can be 1 hour where each operation takes 1 ns of time would be given by $n = 1,897,366$. But this can only be true in the ideal scenario where the machine is only running this one task and there aren't any background tasks taking up computation time and where each operation takes 1 ns.