Arrays and performance

Nariman haidar

Spring Fall 2022

Introduction

In this assignment we going to focus on algorithm performance and that can be with time efficiency of different operations over an array of elements.

We going to performance measurement, presenting numbers and realizing that there is a fundamental difference between what is called O(1), O(n) and O(n2). That if we get time complexity O(1) so that means that is the best algorithm that we have. If we get time complexity O(log(n)) so that means that is not the best algorithm that we have but it is very OK. If we get time complexity O(n) so that means that is not the best algorithm that we have but it is OK. If we get time complexity O(n2) so that means that is not OK and we should working in this case to get it better. If we get time complexity O(n3) so that means that the algorithm is bad and its take so much time and that we do not wanted that happened and we should work on it to get it better than that.

In this assignment we are going to benchmark three different operations and determine how the execution time differs with the size of the array. That three operations are:

1- Random access : permitting access to stored data in any order the user desires an array.

2-Search : searching an array means to find a particular element in the array. The search can be used to return the position of the element or check if it exists in the array.

3-Duplicates : it means if we have two arrays and we want to know if we have the same element anywhere in any of two arrays that we have.

Task One:

Random access:

nanoTime:

In the first example there are no operations. which is the time that it takes just only to run the program and it was a little we can see it in the table. And every time I run the code it gives me a different number from the number before it.

i	time
1	200
2	100
3	100
4	100
5	200
6	100
7	100
8	100
9	100
10	100

Table 1: : Output from the first code nanoTime an element

In the second example there is a simple operations like an array and sum so the time that I get here is a little more than the time I get in the previous example and we can see it in the table. In the third example we

i	time
1	300
2	300
3	200
4	200
5	200
6	100
7	200
8	200
9	200
10	100

Table 2: : Output from the second code nanoTime an element

have an array and we defined a type of parameter but we didn't perform any operations after it so it gave a little time when I run the program and it is less than the second example. But there is a note here that every time I run the same code, I notice that it takes a time not similar result that it took when we run it once before, and this I think from my point of view that it depends on the computer that I own, on the operating system I am working on, and on how many processes I run and includes mostly caches and branch predictors.

more operations:

I used the code in this task with the additions that the student must add in order to be able to measure the time by means of the function to measure the time, where in this task we will enter the matrix measurement. Conclusion I found that the time to access a random element in an array decreases when the arrays size increases. And that i don't expected because it expected that every time I increase the size of array then the time increase not the opposite. And every time I run the program it gave me another result not the same result and I don't know the reason behind that but I think there is some thing related to the computer itself and the operative system that running on this laptop and memory also can be effected and the operations that the laptop runs at the same time.

n(array size)	**time**
10	0.85
100	0.63
1000	0.62
10000	0.55

Table 3: : Output from the first code access an element

Task Two:

Search for an item:

I used the code in this task 2 with the additions that the student must add in order to be able to measure the time to a simple search algorithm and see how the execution time varies with the size of the array. There are two codes in the task 2 and for each of them we should choose k (number of rounds) and m (number of search operation in each round) to something that gives you predictable results. I did this as shown in the table 4 and table 5.

In the table 4 we can see that when it was an array with a Small size and the same number of rounds and the same number of search operation in each round so it does not take time less than when the bigger size of the array. So that was not necessary that if the size of an array is small so it will take less time than the larger array. And it is the same in the table 5. The complexity to the first code is f(n) = n*n*n so this is O(n3) and it is the same to the code 2 in this task.

Task Three:

Search for duplicates:

I wrote the code with the same criteria as written in this task. I did as before an run the benchmark for a growing size of the arrays, n. What i did in this task that I wrote the code through doing a loop over two arrays and comparing each element to every other element. For doing this, we are

n(array Size)	**time to code1**	**k (number of rounds)**	**m(number of search)**
10	1.35	10000	100
100	5.6	10000	100
1000	6.8	10000	100
10000	8.9	10000	100

Table 4: : Output from the first code search an element

n(array Size)	**time to code2**	**k (number of rounds)**	**m(number of search)**
10	3.4	10000	100
100	6.4	10000	100
1000	5.3	10000	100
10000	4.9	10000	100

Table 5: : Output from the second code search an element

using two loops, inner loop, and outer loop. The complexity to this code is f(n) = n + n +n2 so this is O(n2).

n(array size)	**time**
10	1600
100	8280
1000	4285
10000	59700

Table 6: : Output from the code Search for duplicates

In this task when size of arrays are small so time of execution are small and every time the size of array is bigger so the time of execution is bigger.