Arrays and performance

M

VT 2026

Introduction

This report presents the introductory assignment for the Data Structures and Algorithms (ID1021) course.

The objective of this report is to explore the efficiency of different operations over an array of elements. We will benchmark three different operations and determine how the execution time differs with the size of the array.

These Operations are:

- **Random access:** reading or writing a value at a random location in an array.
- **Search:** searching through an array looking for an item.
- **Duplicates:** finding all duplicates in an array.

Clock Accuracy

We were asked to first compile this code and see what the output will be and then based on the outputs we have to come up with a conclusion.

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/time.h>
#include <time.h>

long nano_seconds(struct timespec *t_start, struct timespec
↪  *t_stop) {
    return (t_stop->tv_nsec - t_start->tv_nsec) +
           (t_stop->tv_sec - t_start->tv_sec)*1000000000;
}

int main() {
```

```
    struct timespec t_start, t_stop;
    for(int i = 0; i < 10; i++) {
        clock_gettime(CLOCK_MONOTONIC, &t_start);
        clock_gettime(CLOCK_MONOTONIC, &t_stop);
        long wall = nano_seconds(&t_start, &t_stop);
        printf("%ld ns\n", wall);
    }
}
```

I have tried this code several times, but the outputs were the same each try. **The following are the outputs of the code above:**

Run	1	2	3	4	5	6	7	8	9	10
Result	0 ns	0 ns	0 ns	0 ns	0 ns	0 ns	0 ns	0 ns	0 ns	0 ns

Table 1: Output 1

The Result indicates that all of the outputs were 0 ns. The **reason** why this happened is that the clock is not accurate enough to measure the very fast operations. One operation happens faster than the clock can capture.

This demonstrates that measuring a single array access is unreliable therefore we must benchmark multiple operations together.

```
int main() {
    struct timespec t_start, t_stop;
    int given[] = {1,2,3,4,5,6,7,8,9,0};
    int sum = 0;
    for(int i = 0; i < 10; i++) {
        clock_gettime(CLOCK_MONOTONIC, &t_start);
        sum += given[i];
        clock_gettime(CLOCK_MONOTONIC, &t_stop);
        long wall = nano_seconds(&t_start, &t_stop);
        printf("one operation in %ld ns\n", wall);
    }
}
```

Run	1	2	3	4	5	6	7	8	9	10
Result	0 ns	0 ns	0 ns	0 ns	0 ns	0 ns	0 ns	0 ns	0 ns	0 ns

Table 2: Single array access

This indicates that capturing a single array access resulted in 0ns across all 10 iterations. This occurred because the operation was too fast to measure.

Random Access

```
int sizes[] = {1000, 2000, 4000, 8000, 16000, 32000};
int loop = 1000;
int k = 10;
for (int s = 0; s < 6; s++) {
    int n = sizes[s];
    long min = LONG_MAX;
    for (int i = 0; i < k; i++) {
        long wall = bench(n, loop);
        if (wall < min)
            min = wall;
    }
    printf("%d %0.2f ns\n", n, (double) min/loop);
}
return 0;
```

The code above implements a benchmark for measuring random array access. It takes 1000 values randomly picked from an n sized array. After accessing the array 1000 times it then divides the printed value by 1000.

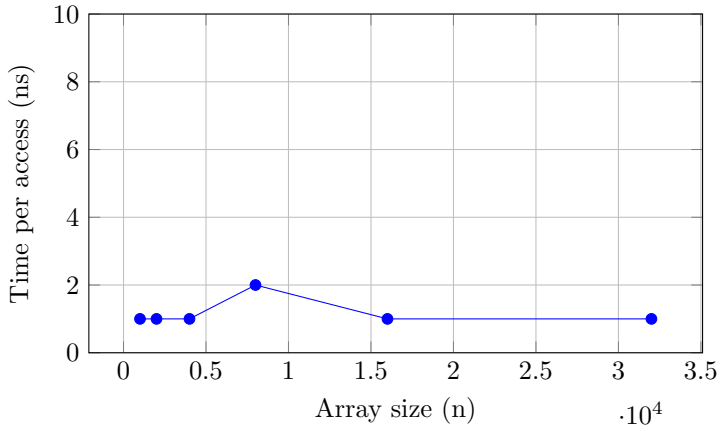

Figure 1: Array size impact on random access execution time

Figure 1 clearly shows that the time per access remains nearly constant as the array size increases which confirms that random array access has constant time complexity $\mathcal{O}(1)$.

Search for an Item

Now, we will measure the runtime of the search algorithm on arrays of varying sizes.

```
long search(int n, int loop) {
    int *array = (int*)malloc(n*sizeof(int));
    for (int i = 0; i < n; i++) array[i] = rand()%(n*2);
    int *keys = (int*)malloc(loop*sizeof(int));
    for (int i = 0; i < loop; i++) keys[i] = rand()%(n*2);
    int sum = 0;
    clock_gettime(CLOCK_MONOTONIC, &t_start);
    for (int i = 0; i < loop; i++) {
        int key = keys[i];
        for (int j = 0; j < n; j++) {
            if (key == array[j]) {
            sum++;
            break;
            }
        }
    }
    clock_gettime(CLOCK_MONOTONIC, &t_stop);
    long wall = nano_seconds(&t_start, &t_stop);
    return wall;
}
```

The code above here is the implementation for method of search algorithm and it searches for a specific item varying the size of array.

As seen in Figure 2, when the array size increases, the runtime increases linearly. The time complexity for the search algorithm can be written as $\mathcal{O}(n)$.

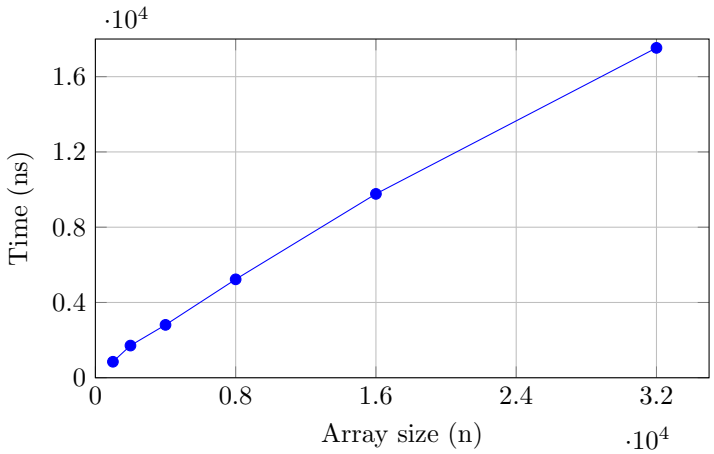

Figure 2: Search algorithm time performance

Search for Duplicates

The final task in this assignment was to implement the duplicate detection algorithm and measure its runtime performance.

```
long duplicates(int n) {
    int *array = (int*)malloc(n*sizeof(int));
    for (int i = 0; i < n; i++) array[i] = rand()%(n*2);
    int sum = 0;
    clock_gettime(CLOCK_MONOTONIC, &t_start);
    for (int i = 0; i < n; i++) {
        int key = array[i];
        for (int j = i+1; j < n; j++) {
            if (key == array[j]) {
                sum++;
                break;
            }
        }
    }
```

The Search Algorithm compares each element with every other element in the array using two nested for loops.

Looking at Figure 3 when the array size doubles, the runtime increases by almost four times. Therefore the time complexity for the duplicate detection algorithm can be written as $\mathcal{O}(n^2)$.

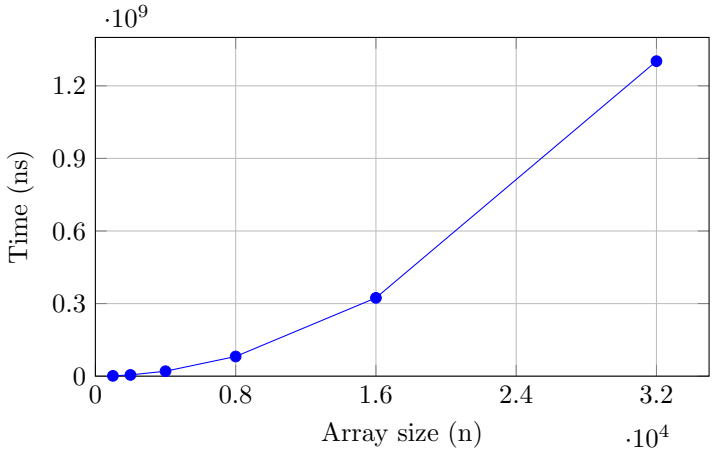

Figure 3: Duplicate detection algorithm time performance