CS 4080 Final Project Report

Team: Home United
Jennifer Chiang
Rachel Lewis
Mai Luu
Sofia Pineda

Table of Contents

*This is just for easier traversal and access to different part of the doc we can take it out later

Abstract

Array handling is crucial in the development of programmers however it also differs in the different programming languages. This project seeks to differentiate the differences of array handling in four different languages: Java, Javascript, C++, and Python. Based on the results of array addition between one-dimensional and multi-dimensional arrays Java had an extremely close time, Javascript and C++ had multi-dimensional arrays performing better than one-dimensional arrays, whereas it was the exact opposite for Python. *Someone pls add the test 2 overarching conclusion thanks*

Introduction

Arrays are an important aspect in every programming language and it is one of the most common types of data storage and almost every modern programming language uses arrays in some way. This is because arrays are needed to perform quick operations on large sets of data. There are certain requirements that the arrays, in any programming language, must meet to

accomplish those goals. Creating an array within a program or object's scope should be easy and fast, read and write operations should be quick, and the array's data and contents should be volatile so that when a program terminates, it's associated arrays are terminated too so that memory is freed up.

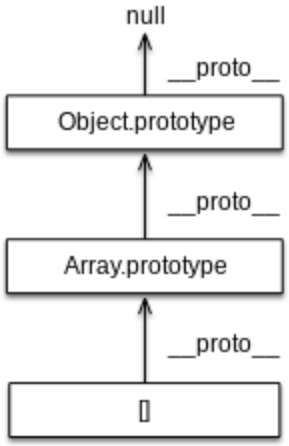
This leads to the main objective behind this project, which is to study whether the arrays in Java, Javascript, C++, and Python meet the stated requirements. We will also compare arrays between the four languages by researching how design decisions impact an array's effectiveness as a data structure for each language.

Ultimately, these major design decisions have an enormous impact on an array's performance and use. The way an array is implemented in any language changes the kinds of operations you can perform on it, such as matrix, arithmetic, creation, copying, and slicing operations. Some of these operations are not possible at all depending on the array's implementation within a given language. Our project, including this report, will dive deep into these design issues to explore the strengths and limitations of arrays of each language while evaluating their overall effectiveness as a data structure.

Methods/Algorithms/Concepts

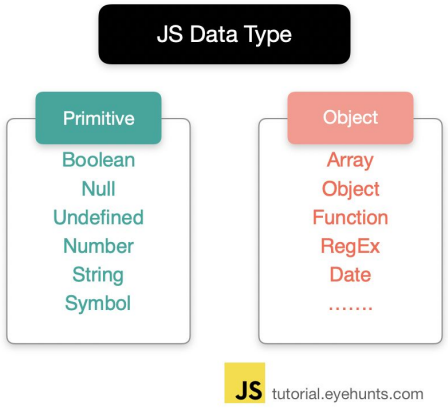
Inheritance

The way inheritance works for all objects in Javascript, including arrays, is with what is called prototype-chaining. Based on the official MDN web docs on inheritance in Javascript, all objects are instances of `Object` and thus inherit all of `Object`'s properties. Within every object, there exists a private property which holds a link to another object called its prototype. This allows for objects to call the functions of it's prototype, the functions of the prototype's prototype, and so on. Thus creating a long chain of prototypes, with each one being able to access



the properties of the prototype above it. According to the MDN web docs on Arrays, all arrays in Javascript inherit it's functions from `Array.prototype`, which inherits it's functions from `Object.prototype`. Javascript arrays are able to access and use the functions in both the `Array` and `Object` prototypes.

Data Types

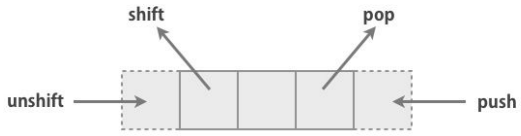


In Javascript, neither an array's length nor the types of it's elements are fixed. According to the Javascript MDN docs, Javascript allows for the data type of any variable to change at any time. Thus, variables in Javascript could be of any data type. Objects could also be of any data type and since arrays in Javascript are objects themselves they, too, could be of any type. This allows for an array to contain objects of possibly any type. Because of this, Javascript does not perform type-checking, since such an implementation would be useless. What would be the point of checking an element's data type before adding it to an array if that type will change later on in the code? More freedom is given to the programmer to be able to store many different types of data into an array without being constrained to checking if all the elements being added are of the correct type, at the cost of burdening the programmer for checking such a condition manually if separation between data types is important to them.

Static vs. Dynamic Sizing

All arrays in Javascript are dynamically sized. This is due to the properties of the `Object` prototype, which all `Array` prototypes inherit from. The `Object.prototype`'s length value is never a fixed value and it is allowed to constantly change. Another unique feature of array sizing in Javascript is that arrays also inherit the functions `pop()`, `push()`, `shift()`, and `unshift()`

from the `Array` prototype. These allow for elements to be freely removed or added to the array during a script's execution. After any one of these operations, the length property will always be changed, thus length is a dynamic value. The `pop()` and `push()` functions mimic those of the stack methods of the same name, whereby elements will only be added to or removed from the array at index `[(length--)-1]`. The `shift()` and `unshift()` functions perform the same operation as the `pop()` and `push()` functions except they are slower, since `shift()` and `unshift()` only remove or add zero as the first element, respectively, while the indices of all other preceding elements have to be decremented or incremented.



Due to this feature of Javascript array sizes, there is no such thing as `IndexOutOfBounds` errors, since if an element is placed at a positive index that doesn't exist, then Javascript will dynamically resize the array. The array's `length` property will be changed to `(y+1)`, where y equals the index of the recently added element.

Arrays in C++ are divided into two categories: static and dynamic arrays. The memory of static arrays are allocated at compile time and it can not be modified after. The dynamic array is where its memory is dynamic allocated using pointer variables and memory management functions such as malloc, calloc, and realloc. For a dynamic array, the length of the array is not known at compile time, and its size can be automatically resized if new elements are added or deleted. A simple dynamic array is constructed by allocating an array of fixed-size, larger than the number of elements immediately required.

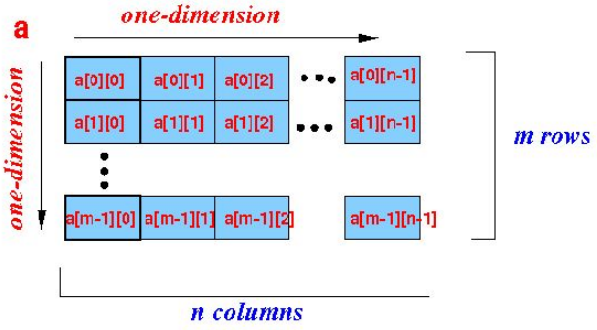
Java arrays are very similar to C++ arrays with a few key differences. One simple difference is that C++'s built-in method for calculating the length of the array uses sizeof() while Java uses length(). All arrays in java are dynamically allocated meaning that Java only declares a reference to an array but the storage is not yet allocated until runtime.

In addition, Java has many built in functions to manipulate arrays and assist the programmer. These functions can be found in java.util.Arrays and by calling Arrays.function(). For example, within these functions are a Binary Search, a copier, and a filler.

Unlike the other languages Python has no data structure named “arrays”. In fact in the absence of arrays Python has data structures called: lists, tuples, dictionaries, and sets. When common attributes of arrays are examined it almost always includes indices, an ability to call upon elements through the index, and mutability. After considering all three criterias the data structure that most closely resembles arrays in Python are lists. Python lists are dynamically sized and new objects can be added to the list through the built in list methods. Lists in Python are dynamically sized whereas tuples are static and elements in a tuple can never be changed. On top of being dynamically sized arrays in Python are also able to accept heterogenous types for its elements. An array does not have to be an array of only integers, in Python the array can be made up of strings, integers, and even dictionaries just to name a few.

Multidimensional Arrays & Matrix/Vector Operations

Most matrices and vectors that are created in Javascript are implemented with a 2D-array, also known as an “array of arrays”. This implementation is quite simple, in that all it involves is creating a standard array in Javascript and filling it with more arrays. The “inner” arrays that are placed in the “outer” array must all be the same size in order for it to truly count as a matrix. The length property of the outer matrix equals the number of rows and the length property of all the inner arrays equals the number of columns. Vectors work a little differently, in that they are matrices with one column but many rows. Vectors in Javascript are implemented using the same array-of-arrays method except the length property of all the inner arrays has to equal 1.



One of the biggest limiting factors to matrices in Javascript is that linear algebra, and operations which are classically associated with linear algebra, is not

possible on such arrays. This is because all arrays in Javascript, and all matrices formed from those arrays, are one-dimensional. Even so-called “multidimensional arrays” are not truly multidimensional, since they are composed of a one-dimensional array of one-dimensional arrays. Thus, matrices in Javascript don’t meet the definition of a real and true matrix, so operations like matrix multiplication, matrix addition, vector spaces, determinants, eigenvalues, eigenvectors, and transformations are not possible for Javascript matrices.

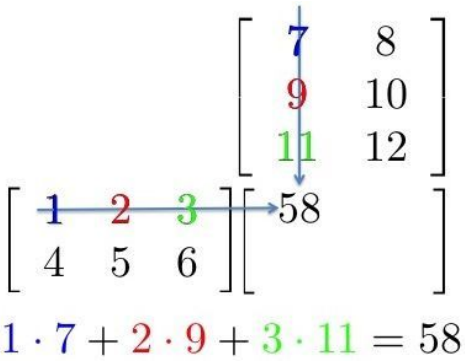
In the field of linear algebra a real and true matrix, if implemented correctly in a programming language, would take a form more akin to one array whose dimensions would be 1x1. Within that array would **not** be more arrays, but there would actually be multidimensional space just in that one index, in that one array. In mathematics, this is essentially what a matrix really is, which is a set of linear equations whose columns contain all the dimensions that span all those equations and whose rows contain all the equations that span the matrix’s vector space. Because of this definition, a matrix only represents one set of linear equations. This definition of matrices does not match the implementation of them in Javascript, which mainly utilizes a set-of-sets method to create matrices.

$$A = \begin{matrix} \\ \\ \\ \\ \\ m \text{ rows} \end{matrix} \overbrace{\begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{i1} & a_{i2} & a_{i3} & \dots & a_{in} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \dots & a_{mn} \end{bmatrix}}^{n \text{ columns}}$$

There is one way around these limitations for matrices in Javascript, which is through the Math.js library. Math.js has to be installed first with npm and then it must be loaded and instantiated before it can be used in code. With the Math.js library, matrices can be created and more arithmetic operations become available to you while using this library such as adding, subtracting, multiplying, and dividing two matrices. The determinant of a matrix can be found using `math.det()` as well as its factorial, with `math.factorial()`. However, despite all the Math prototype’s functions being available to use by this special kind of array in Javascript, it

still doesn't meet the true definition of a matrix and thus matrix-specific operations are still limited and the operations that are available are technically not real matrix operations.

For a brief example, in linear algebra, two matrices can never be multiplied together if the number of matrix A's columns doesn't match the number of matrix B's rows. However, with the Math.js library, this condition is completely ignored and the operation is performed anyways. Careful examination of this specific operation shows that the Math.js matrix multiplication method is of the form:

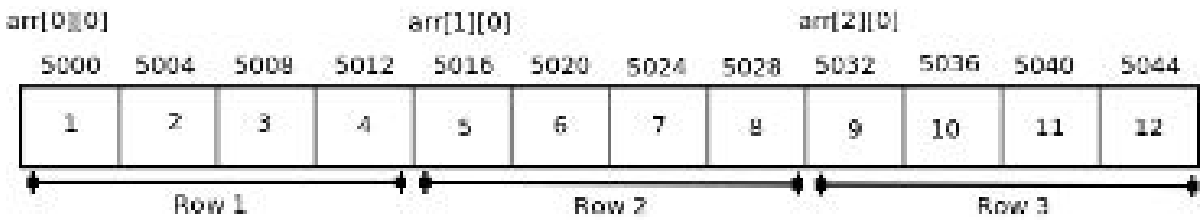


$$\sum_{n,m=1}^{n,m} (MatrixA[m][n] * MatrixB[m][n])$$

This form is not correct compared to the mathematical definition of matrix multiplication, whose form a bit more complicated:

$\sum_{k=1}^{k} (\sum_{m=1}^{m} (MatrixA[m][(1,n)] * MatrixB[(1,j)][k]))$, where (m x n) are the dimensions of MatrixA and (j x k) are the dimensions of MatrixB. Because of these differences in the matrix multiplication method between the two implementations, this shows that all the matrix operations that are associated with the Math.js library (not just multiplication) are superficial, which displays how much the general form of matrices in Javascript differs from that of real matrices.

Multidimensional arrays in C + + are described as an array of arrays. For example, a two dimensional array is an array of one dimensional arrays, so if the array is m x n dimensions, which represents 'm' number of 1D arrays with 'n' elements inside. The concept of storing 2D arrays in the memory is that it is store in row-major order, so rows are placed next to each other. Each row is considered as a single dimensional array.



In the picture above, we can see that the element at first row and first column is being stored at the memory address of 5000, and the next element of the first row is stored next to it. Then, the first element of the second row is stored right next to the last element of the first row, and this pattern is repeated until the element of the last row and last column is stored.

Same concept for a three dimensional array, they are the array of two dimensional arrays. For example, an array[2][3][4] represents 2 arrays of 3 x 4 array.

There are some different concepts of multidimensional arrays in C++, specifically two-dimensional arrays:

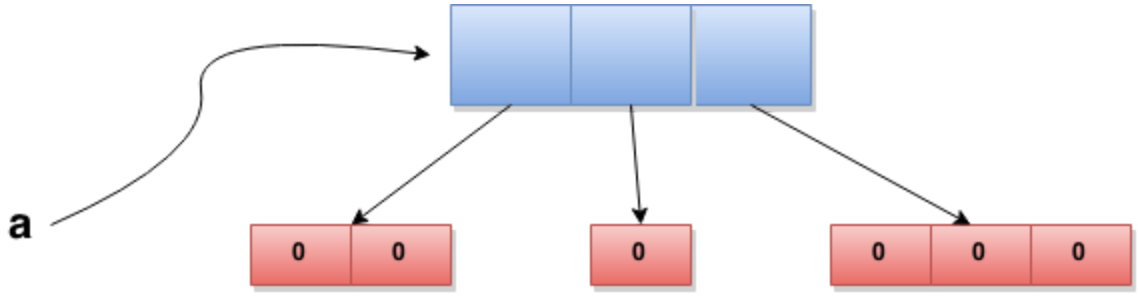
- The two-dimensional array: int arr[][]. It cannot be resized in any direction and is contiguous. Must be allocated statically.
- The pointer-to array: int (*arr)[]. It can be resized only to add more rows and is contiguous. Must be allocated dynamically but can be freed free(x).
- The pointer-to-pointer: int **arr. It can be resized in any direction and is not necessarily square. Usually allocated dynamically and freeing is dependent on its construction.
- The array-of-pointers: int *arr[]. It can be resized only to add more columns and is not necessarily square. Resizing and freeing also depends on construction.

Much like C++, in Java, multidimensional arrays are represented as an array of arrays.

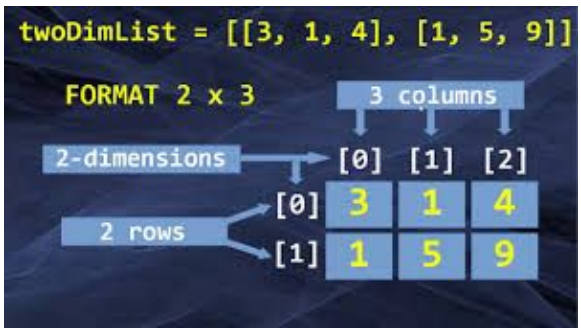
	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

This picture describes how one would visualize and accurately read a two-dimensional array in Java. When declaring an array like so-- int[][]-- this indicates that there will be one array and then that array is composed of other arrays. The dimensions are determined by how many levels of arrays there are. In the example above, there are two levels of arrays and thus two-dimensions. Thus, a 3D array would be declared with int[][][].

In memory, the two-dimensional array would look like the following image:



In Python multidimensional lists are built as lists within lists. Like one dimensional lists multidimensional lists are also dynamic and additional data can be added to the list easily through the language's built in methods. When the len method is called for multidimensional lists it returns the total number of elements within the list, the len method does not return the number of rows and columns within the array.



Due to the characteristics of single and multiple dimensional arrays, its applications have been used widely in many subjects and important algorithms. One of the applications is the matrix operation such as adding, subtracting, and multiplying as mentioned above. Two dimensional arrays are also applied in some algorithms to find the shortest path such as Floyd's, Djikstras, and so on. A specific example for three dimensional arrays is a searching algorithm based on three resources. For example, searching a word in a dictionary with the given page number, line number, and column in which the words belong.

Array-Specific Methods

In Javascript, all arrays inherit their functions from the Array prototype. For the sake of brevity, not all of the functions that are available to Javascript arrays will be discussed in this section (there are 33 functions in total). Instead, what will be discussed are some array methods that are unique to Javascript or are implemented across all languages' arrays but implemented differently in Javascript.

One of the most classic methods for arrays in any language is the length or size method. These methods are crucial when it comes to data handling, validation, and iteration over arrays. Javascript does offer a means of accessing the size of the array's length, which is the number of elements in the array. In Javascript, `.length` is actually not a function or method, but it is a property of the array. Unlike compiled languages, Javascript's length property is writable, meaning that it can change. This feature of the `.length` property is a remnant of Javascript's dynamic array sizing. Because of this dynamic sizing for Javascript arrays, it is also possible to add or remove items freely from the array during the script's execution. This is done via the `pop()` and `push()` functions. The `push()` function will add an element to the end of the array and return the new length size of said array. `pop()` will remove an element from the end of the array and return said element. Two very useful functions for quickly mutating an array in Javascript without having to give a specific index.

Another very useful function that is unique to Javascript arrays is the `.find()` function. This function iterates across the array and returns the first value it finds that meets some kind of condition given by the programmer. This saves time and lines of code for the programmer, since in most programming languages this kind of function is only accomplished by iterating across the array in a loop, checking each element against a condition, and then returning a boolean. The `.find()` function does all of this for you all on one line. Javascript arrays can also have their elements sorted in ascending order using the `.sort()` function. Although this function is not unique to Javascript (Java also has a sort method for it's arrays), it is more powerful in that it can also sort an array of strings in alphabetical order. The Java `sort()` method for arrays only has constructors for arrays of all data types except strings and booleans. This, again, is very handy for any programmer who wishes to sort an array, on one line of code, without having to implement their own sorting algorithm, like for other languages.

Javascript offers a plethora of other mutator and iterator methods for it's arrays, most of which are not outlined in this report because there are so many! `join()`, `keys()`, `map()`, and `filter()` are also some other interesting and useful methods for Javascript arrays that allows the programmer to perform one operation on or apply a condition to an entire array using built-in functions that are native to the `Array.prototype` object. The best part is most of those functions are unique to Javascript arrays since they are not implemented natively in C++, Javascript, or Python.

Array pointer in C++, which can be used to access the first element of an array, or points to the whole array. It can also access the element in multiple dimensional arrays.
We can create the pointer that:

- Points to the first element of array (0^{th} index)

```
int *p;
int arr[10];

p = arr;  // points to the 0th index element of array
```

- Points to the whole array.

```
int(*ptr) [10];
int arr[10];
ptr = &arr;    //points to the whole array
```

We can also use pointer to access the element in 2D array `arr[i][j]` by using the expression `*(*(arr + i) + j)`:

- `*(arr + i)`: pointer to the i^{th} element of arr -> points to i^{th} element of 1-D array. Then the expression `*(arr + i)` gives us the based address of i^{th} element of 1-D array, which gives the same address as arr[i].
- `*(arr + i)` and `arr[i]` are based address of i^{th} 1-D array, it points to 0^{th} element of i^{th} 1-D array
- Similarly, *(arr + i) + j points to the value of j^{th} element of i^{th} 1-D array. It points to the same value as `arr[i][j]`.

With the same concept, we can use pointer to access elements in 3D array `arr[i][j][k]` by using `*(*(*(arr + i) + j) + k)`

As mentioned earlier, Java has many built-in methods that can assist the programmer. Most of these utilities are found in java.util.Arrays. One of these key features is a built-in binarySearch() function that can input an array and a value, returning the location of the value in that array. Variations include only searching from certain indices if that programmer wants to. The compare() function will see if two arrays are equal and contain the same elements in the same order. CopyOf() will copy a specified array into another array. If the two arrays are not the same size, then Java will truncate or pad with zeros so as to prevent fatal errors. One could even copy using a specific range with copyOfRange(). Fill() will fill an array with a specified value and use also be used with specific indices. ParallelSort() will sort the array in whichever order the user specifies: ascending or descending. This is different from sort() but both will get the job done. The parallelSort() breaks the array into sub-arrays that are themselves sorted. This is a common divide and conquer method of sorting. According to the documentation, sort() uses a Dual-Pivot Quicksort.

As mentioned before in the previous sections, Python has many helpful built in methods to make the programmer's life much simpler. Some methods were already introduced and touched upon in previous sections such as len(), the method returns the length of the array. Through the use of the len method, programmers can quickly find out how many elements are in the array. In order to add elements to the dynamic arrays known as lists in Python the append method is used. When called upon the method takes in an element to be appended to the array and adds the new element to the end of the array. There are many more built in methods for the data structure list such as clear, pop, copy I simply mentioned the two most commonly used methods in detail.

Experiments

In order to learn more about arrays in these languages, two experiments were performed. One experiment is composed of adding two very large arrays together and timing the operation. Comparisons were made between one-dimensional arrays and two-dimensional arrays. The second experiment was simply performing a binary search on large arrays, arrays of 5 different input sizes were used to gauge whether array size affects program read and write times. For both experiments, a time snapshot was taken both before and after their respective operations took place, mainly before they entered a while loop and after they left it. The difference was taken between the two time snapshots and the results of those were summed and then averaged to make conclusions about the data. The operation was performed at least ten times in order to receive an average for the average runtime.

Results

Java

In the first experiment, we fill two arrays with the values 5 and 2 respectively. After making the result array, we begin the timer. The for loop adds both arrays into the result array

and once that operation is finished, the timer stops. The average recorded for this experiment clocked in at 15,492,140 nanoseconds or 15.49214 milliseconds.

The second experiment is largely the same, except we use the two-dimensional arrays. The average recorded for this experiment clocked in at 15,617,720 nanoseconds or 15.61772 milliseconds.

The two averages that we computed were very similar, meaning that Java handles multi-dimensional arrays as efficiently as it handles single-dimension arrays of equal size. This is significant because the same experiment in other languages such as python or JavaScript yields a significant enough difference in time when using arrays of varying dimensions.

Finally, the last experiment involves using a Binary Search on a sorted array of varying sizes. Here is the table of results:

Array Size	Time for Search
3,000,000	26,900 ns
15,000,000	18,800 ns
7,500,000	13,300 ns
3,750,000	4,800 ns
1,875,000	2,700 ns

When the array is significantly larger, the time of the binary search will take longer. As seen with the first two arrays, when the array is double the other, the time will also approximately double. The time is directly proportional to the size of the array.

C++

- First Experient is performing the addition between two single dimensional arrays with the size range from 100 – 250,000, and the times (in nanoseconds) are presented in the table below:

Size	100	10,000	40,000	80,000	100,000	160,000	250,000
Time	695	48,953	293,957	480,584	668,793	697,548	1,627,071

o Second Experiment is performing the addition between two two dimensional arrays with the size also range from 100 – 250,000, and the times (in nanoseconds) are presented below:

Size	100	10,000	40,000	80,000	100,000	160,000	250,000
Time	818	64,635	349,308	321,034	511,943	627,390	1,250,464

After finishing these two experiments and comparing the numbers from the tables:

- The results show that the performance of adding two single dimension arrays for the size below 50,000 are faster than adding two 2D arrays.
- However, when the size reaches 50,000 the performance of adding two multidimensional arrays are faster than adding two single dimension arrays.
- Since the array is stored as an array of arrays in C++, and the pointers are pointed to other arrays, so the performance of adding 2D arrays are faster than 1D arrays when the size grows bigger.

o Third Experiment is doing binary search on the various sizes of sorted single dimensional array. The table below is the time (in nanoseconds):

Size	100	1000	10,000	50,000	100,000	150,000	500,000	1,000,000
Time	3045	4109	4873	4472	5803	5465	6839	8287

Result for this experiment: as the array size goes bigger but the time for searching a value through a binary search algorithm is nearly constant with regard to the size, this matches with the original expectation.

Python

```
Average of 10 runs for one-dimensional array: 6068685010.0 nanoseconds
Average of 10 runs for multi-dimensional array: 8080533330.0 nanoseconds
```

For the first experiment, although the arrays of the same dimension and data were added together the multi-dimensional array dimension took noticeably longer to perform the same task. In fact, a difference of 2,011,848,320 nanoseconds was found (see equation 1). A percent difference of approximately 33% (see equation 2) was found when comparing the multi-dimensional array runtime and one-dimensional array runtime of addition. Despite the arrays having the exact same size and data, the multi-dimensional array took longer to perform the same task. Despite the arrays having the exact same size and data, the multi-dimensional array took longer to perform the same task. This is probably due to the implementation and where the time was started. Since python stores multi-dimensional arrays as a list within a list it should store the multidimensional array as two one dimensional arrays of size 5477. However due to where the start time started recording, the two for loops in order to access the elements in the multidimensional array is also taken into account for the calculation of the runtime. Based on the placement of the timestamps it is no wonder that the results reveal that multidimensional arrays take longer to run. This test should be revised and further researched as a possible future work topic.

Equation 1: $8,080,533,330 - 6,068,686,010 = 2,011,848,320\ nanoseconds$

Equation 2: $\left|\frac{8,080,533,330-6,068,686,010}{6,068,686,010}\right| * 100 \approx 33.15\%$

In terms of the second test, despite going up to 30 million the runtime for binary search continued to have an average runtime of zero. Bigger input sizes may have been needed. However, the creation of bigger arrays may cause a memory error, additional tests may be done in order to further investigate. I see this as a possibility as a future work topic. The binary search

algorithm was checked against various code sources numerous times, however it could still be possible that due to a personal bias I failed to catch an error in the implementation.

Javascript

The results for the first Javascript experiment were surprising. The initial purpose of creating two test files, one for arrays and one for 2D arrays, was to see if an array's dimension affects the speed at which operations can be performed on it. The initial hypothesis was that performing operations on 2D arrays would take longer than for 1D arrays. However, the results of the experiment show the opposite, that operations on 1D arrays are slower than for 2D arrays, even if both arrays are identical in size. This experiment showed that the average amount of time it takes to add two 1D arrays of size 29,997,529 is ~3.421 ms and the average time per operation between two elements in both arrays ~1.141e-7 ms. For test2.html, the average amount of time to add two 2D arrays of size 5,477x5,477 was ~0.051 ms and average time per operation was ~1.702e-9 ms. In conclusion, it takes ~67x longer to add two 1D arrays than it takes to add two 2D arrays in Javascript. It also took 67x longer to perform addition between two elements in a 1D array as opposed to a 2D array.

The results of the second experiment were predictable and expected. The experiment proved that as an array gets larger, then the time it takes to read values from that array and perform a search on it also increases. More specifically, the equation that best represents the association between array size vs. search time is $-6.73 * 10^{-18}x^2 + 3.92 * 10^{-10}x + 3.37 * 10^{-3}$. This equation is of polynomial form because there is roughly an x^2 correlation between array size and search time. It appears that after the array size exceeds 25 million, there is not much change in average search time, which is a little surprising. My initial hypothesis was that array size vs. search time was logarithmic and as array size increases, the search time also increases exponentially. But the graph of the data clearly shows that average search time plateaus after the array size exceeds 25 million. The R^2 of the equation (the degree to which the equation

matches the data) is 0.945, which is very good. It shows that ~95% of the data matches the equation given by the curve.

Future Work

In regards to the future the possibilities are endless, however there are a couple experiments that immediately come to mind as a possible continuation of this project. The emphasis of this particular project was on how arrays are implemented in the four different languages: Java, C++, Javascript, and Python, however the experiments done is not enough to encompass all that arrays have to offer. Since experiment two was based on doing binary search on one dimensional arrays and comparing the runtimes based on the input sizes, it would not be a stretch to further investigate the search algorithm's effectiveness on multidimensional arrays. On top of that, a project specifically on utilizing the built in array methods would also be very interesting as these languages have their own unique methods. These are just some ideas for future projects to pursue based on the experiments done within this project.

Conclusion

Based on the experiments performed, the fact that compiled languages tend to run faster than interpreted languages is confirmed as the runtime for Java and C++ tended to be faster than that of Python and Javascript overall. In terms of array handling based on dimensions, Java handled one dimensional and multidimensional arrays on similar standards. On the other hand, Javascript and C++ had multidimensional arrays performing better than one dimensional arrays overall. Finally in terms of Python one dimensional arrays performed much better than multidimensional arrays. When analyzing the results from the second experiment there were some discrepancies here and there, however the general consensus was that as the size of the input array increased so did the time it took to run the search algorithm which was nice to have

confirmed. The discrepancies that occurred was most likely caused by the trial sizes being too small leading to an unreliable average runtime. The languages each have their strengths and weaknesses hence why it is important to know what the project seeks to accomplish before settling on a particular language.