Dynamic Arrays

By: Jacob Becker

"It is not the strongest of the species that survive, nor the most intelligent, but the one most responsive to change." — *Charles Darwin.* This idea of adaptability applies not only to biology but also to computer science. In programming, data structures that can adapt to changing demands are essential for efficient performance. One such data structure is the dynamic array. Unlike a static array, a dynamic array is a list of data that can grow or shrink as elements are added or removed. The number may be small or large, but data will have to be handled promptly. It is up to us programmers to make this process as fast and efficient as possible. We interact with arrays every day, for instance your camera roll is an array of images, even looking at an image you are interacting with an array. Arrays have been around since the beginning of modern computers themselves. Dynamic Arrays began appearing when the first programming languages were created.

We are only going to focus on appending data to an array. Appending is simply adding data to the end of the array. However, it is important to understand how arrays are stored in computer memory. Let us say we have an array, `A`, and `A` has 10 elements. We can represent those elements like this: `A = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`. In computer memory, this array is stored contiguously, meaning each element is placed next to the previous one in a sequence. This contiguous memory allocation allows for efficient access to elements via indexing. If we want to add the number `11` to the end of the array, the operation might seem straightforward with `A.append(11)`, resulting in `A` looking like `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]`. In high-level languages like Python, this operation is handled automatically. Python's list data structure dynamically resizes and allocates more memory as needed, so we do not need to manually manage this process. In languages like C or C++, appending to an array requires more attention. If the array's allocated size is insufficient, you would need to allocate a larger block of memory, copy the existing elements to this new block, and then deallocate the old memory. This resizing process involves more complex memory management, including handling memory allocation and deallocation manually. Thankfully, in a language like Python, we do not have to worry about these low-level memory management details. Python's built-in list implementation abstracts away these complexities, allowing us to focus on higher-level logic without manually handling memory resizing.

However, we can still make a dynamic array with Python. I would like to note that the built in *list* in Python is implemented in C and will be faster than our dynamic array, but do not let that discourage you if you plan to make a dynamic array module. We still have not really got the grasp of a dynamic array. Let us define capacity as *c* and define *n* as number of elements. The reason this array is “dynamic” is that an array has *n* elements and what a dynamic array does is add capacity, *c*, to this array, or in other words adds some extra space in memory for that element so you do not have to copy and paste each append. As we fill up *n* elements, we want to expand the amount of cushion space by increasing *c* by some amount. This amount can vary some languages, or people, double the capacity some use some other multiple. At the end of the day, it

depends on how many elements will need to be added to the array. I made a Python class that imitates a dynamic array. There is Python code, but I tried to add comments (#) to help you follow.

```
import ctypes # How we can create an array

class DynamicArray(object): # objects mean any value type (int, str, ...)
    def __init__(self, initial_capacity=16):
        self.n = 0 # Number of elements (init with 0)
        self.capacity = initial_capacity # Gives cap 16 as default
        self.A = self.make_array(self.capacity) # Will explain soon
```

The only import we need is `ctypes`. We import this module so we can create arrays that can hold Python objects while interfacing with low-level C-like operations. With classes, we must initiate the object's attributes when a class instance is created. In the `DynamicArray` class, the `__init__` method does just that. We define `self.n` as the number of elements and `self.capacity` as capacity. Now, `self.A` points to a function called `make_array()` and the parameter is `self.capacity`.

```
    def make_array(self, new_cap): # self.A becomes the return value
        return (new_cap * ctypes.py_object)() # returns array
```

This function `make_array` takes in `self.capacity` as `new_cap` and creates an array `new_cap` long. The initial value is 16. Now we must create our append function:

```
    def append(self, ele): # Element is the value you are adding
        if self.n == self.capacity: # Makes sure there is room
            self._resize(int(self.capacity * 2)) # Capacity times 2
        self.A[self.n] = ele # That index now is equal to ele
        self.n += 1 # Adds on total elements
```

This function `append` receives a value, `ele` , entered by the person, as described above. The function then checks if the capacity is full and runs the function `_resize` if `self.n == self.capacity` → `True`. Then we add the value to the Array at element *n*. We also make sure to increment *n* by 1. Now we must explain the final function resize:

```
    def _resize(self, new_cap):
        B = self.make_array(new_cap) # Create a new array with the new cap
        for i in range(self.n): # Copy all elements from the old to new
            B[i] = self.A[i]
        self.A = B # Replace the old array with the new one
        self.capacity = new_cap # Update the capacity
```

This function `_resize` receives the value of `int(self.capacity * 2)`, the `int` is there to make sure the value is not a `float`, or decimal number. `B = self.make_array(new_cap)` is a temporary array. The function copies all elements old to new and array and sets the array to B and updates the capacity. And that is all we need to make an array in Python that can only append items. I also made a slow array that increases the capacity by n + 1 each iteration. This also makes that program extremely slow. The code is the same just `self.capacity + 1` when calling the resize function. You can also find the full code on my GitHub at, https://github.com/JacobB9990/School/tree/main/Dynamic%20Array.

Now, let us compare the two classes. I wrote a simple Python Program that tests both the Dynamic Array and Slow Array. There is a function in this program called that allows me to test multiple times if I would like. Here is that function:

```
def measure_time_for_append(container, max_size: int):
    times: list[float] = []

    for size in range(max_size + 1): # Range: 0-max + 1
        start_time: float = time.perf_counter()
        container.append(1)  # Append an element
        elapsed_time: float = time.perf_counter() - start_time
        times.append(elapsed_time) # Appends to times

    return times # Returns the list of times
```

I am not going to go into detail about this function, but it adds elements to our array and records the time taken for each append operation. Since we already know the total number of elements we are adding, we do not need to store each element's index in a separate list. However, we do need to store the time it takes for each append to analyze the performance. I would also like to note that the timing is per append. We will explore what the graphs will look like if we start the time before the loop later.

Now that we have our function, we can initialize our classes and run the program! I also imported `matplotlib.pyplot` to visualize our data through graphs. What kind of pattern do you think we will observe? Will the data points follow a linear, exponential, or logarithmic trend? Keep in mind that while we can set the maximum range to any value, we are constrained by time, as the slower algorithm takes significantly longer with larger input sizes. I will conduct tests with 1,001 and 10,001 iterations to compare their performance. The machine I am running this code on has an AMD Ryzen 5 5600X with 32 GB of RAM 3200MHz. I used threading to run the Dynamic Array and Slow Array at once just to make the whole program a little faster. Let us run the program.

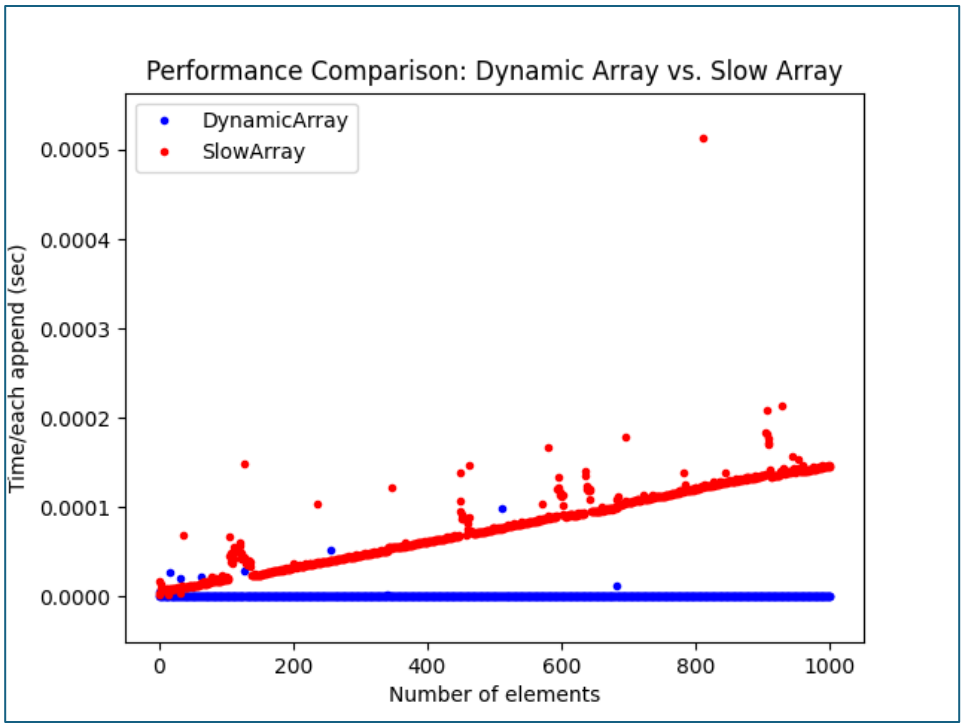

Figure 1. 1,001 appends.

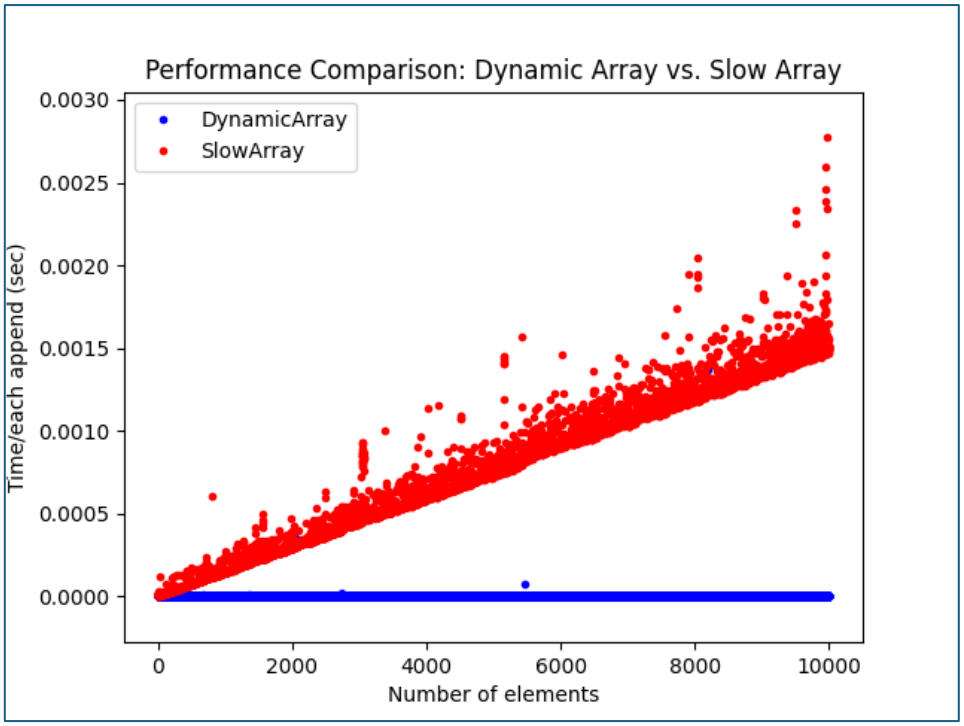

Figure 2. 10,001 appends.

Turns out both graphs follow a linear trend when measuring elements vs. time per each append. Our dynamic array blows the slow array out of the water, just visually alone. In the dynamic array, appending elements is generally fast because resizing happens only occasionally, while the majority of appending elements occur in constant time. This results in a steady linear pattern. However, we do see some blue dots that are slightly off the line, behind some red dots. This deviation happens when we hit capacity limits and need to resize the array. In our case, this resizing occurs at specific points—32, 64, 128, 256, 512, and so on. At each of these points, we need to allocate new memory, copy all the elements from the old array to the new one, and then continue appending. This resizing step takes extra time, which is why those blue dots (representing the dynamic array's append times) slightly spike at those points.

You might be wondering why both Fig. 1 and Fig. 2 show red dots scattered all over the place, representing inconsistent append times in the slow array. This irregularity is due to a couple of factors. The most straightforward reason is frequent memory allocations. Unlike the dynamic array, which increases capacity exponentially, the slow array must allocate new memory and copy all elements every time an element is appended. This process is inefficient and causes the time per append to vary dramatically.

Another potential cause for the erratic pattern is system-level interruptions. When dealing with larger arrays, the operating system’s memory management processes, such as garbage collection or paging, can become more noticeable, resulting in unpredictable slowdowns. Also, other background processes or threads on the system might be competing for resources, causing some append operations to take longer than expected. Lastly, the primary reason behind the scattered red dots is simply that the slow array is, well, slow. Since it only increases capacity by 1 with each append, every single addition involves costly memory reallocations and copying, which makes the append times less consistent.

Now, what if we measure the total time taken from the start rather than timing per each individual append? This gives us a different view of the performance over time. We ran the same experiment with the same number of appends, but this time, we started the clock before the loop and measured the time from that point forward. What would we expect? Will the data points still follow a linear trend, or will we see something exponential or logarithmic? Let us dive into the results.

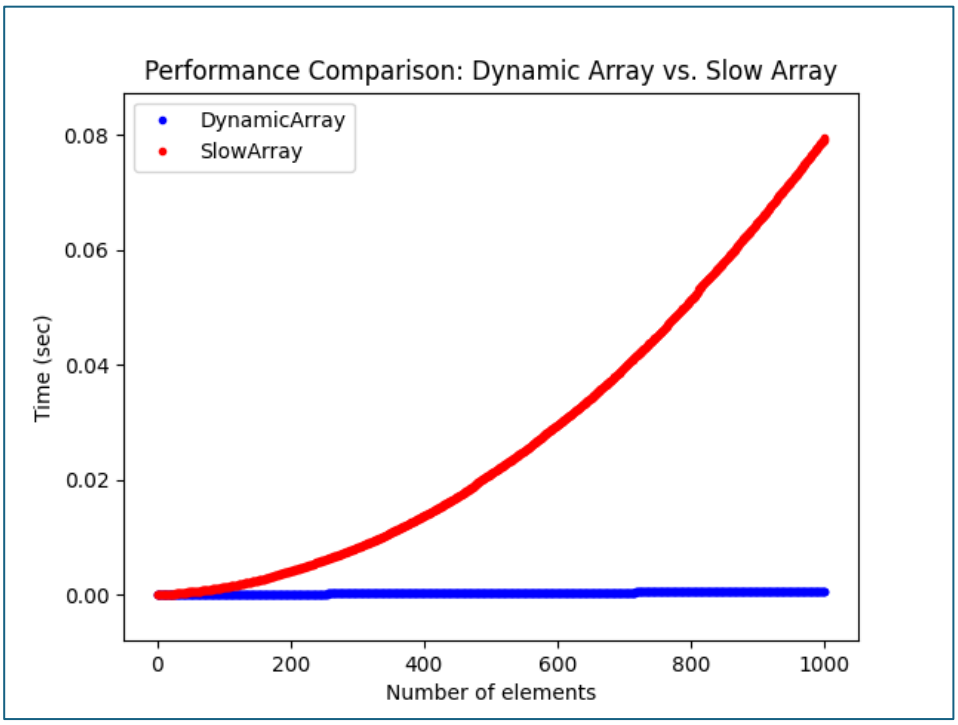

Figure 3. 1,001 appends.

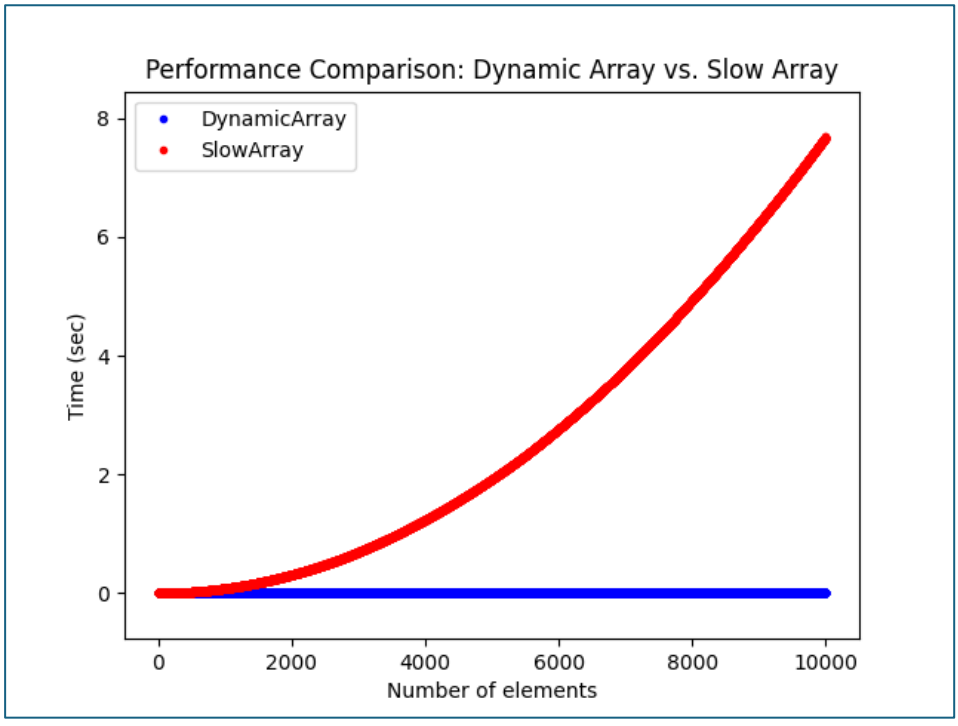

Figure 4. 10,001 appends.

From the graphs above, we can clearly see that the slow array shows a curve. The reason for this curve is straightforward: every time we append an element, the slow array needs to increase its size by exactly 1. This means with each append, the program must allocate new memory and copy over all existing elements. As the array grows, this process becomes more time-consuming, and the time per append increases steadily, creating an exponential relationship between the number of elements and the total time taken. The more elements you append, the more significant this curve becomes. We will dig deeper into the math behind this exponential growth later.

Now, let us talk about the dynamic array. While its performance appears linear, it is not perfectly so. If we zoom in on a segment of, say, 100 elements, we will notice occasional jumps in the time it takes to append. These jumps happen where the dynamic array reaches its capacity limit and must resize. When resizing occurs, the array allocates more memory—usually doubling in size—and copies all the existing elements over. This resizing step causes a brief spike in time, which accounts for those jumps.

However, these spikes become less frequent as the array grows larger. The reason? The dynamic array is expanding its capacity exponentially—doubling each time it resizes—so even though the resizing process is costly, it happens less and less often as the array grows. With enough elements, most append operations occur in constant time, while only a small fraction triggers a resizing. This is why, on average, dynamic arrays are said to have amortized constant time for append operations. Again, we will break down this relationship mathematically later to show why this is the case.

So why do Dynamic Arrays work so efficiently? It might seem that resizing an array would take a lot of time, especially as the array grows larger. However, dynamic arrays are designed to minimize this overhead through a clever technique called amortization. Let me break this down mathematically to show you how, despite occasional costly resizing operations, the overall time per append operation averages out to constant time.

Here is how it works. Imagine we start with an array that can hold n elements, and every time the array reaches its capacity, we double its size. Let us say the initial capacity is 2. If the number of elements is less than the capacity, we can append new elements without issues. However, once we hit the capacity limit, we must double the array's size to make room for more elements and copy all the previous elements. This creates a simple pattern: the capacity grows from 2 to 4, then 8, 16, 32, and so on. Each time the capacity doubles, we make space for more elements, but resizing happens less frequently as the array grows larger. This is what makes a dynamic array run in constant amortized time, $O(1)$. We can apply this formula to this pattern:

$$\mathrm{T}(n) = \mathrm{c} \times \mathrm{n} + \sum_{\mathrm{n}}^{\mathrm{k}} 2^{\mathrm{i}} = \ O(n)$$

Where $T(n)$ is the total time, c is the cost per append (which is constant for non-resizing operations), and the summation represents the cost of copying elements during resizing. When n is small, these time jumps due to resizing are more noticeable. However, as n grows larger—into

the billions, for example—the time spent copying and resizing becomes a small fraction of the total time. This is what justifies the "constant" amortized time complexity per append.

This happens because the summation $\sum_{i=1}^{k} 2^i$ represents a geometric series, and its total sum is proportional to $2n$. Even though resizing requires copying elements, this cost is spread over all n operations, and the number of times we need to resize decreases as the array grows. The overall cost of resizing becomes negligible compared to the total number of operations. Therefore, while individual resizing operations may take longer, the average cost per append remains constant, which justifies the "constant" amortized time complexity per append. In total, the overall time complexity of performing n appends is $O(n)$, and the amortized cost per append is $O(1)$.

Why do we not increase arrays by 1. Well, it is just flat out inefficient. For each append, we must increase the array's capacity which gives us: $O(n)$. Since each append requires more operations than the previous one results in $O(n^2)$. We can see this by using this formula:

$$\mathrm{T(n)} = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

This formula represents an arithmetic series. As said earlier we are copying and pasting each time. We can visually see this in our recorded data. Fig. 1 and Fig. 2 represented the time taken for each append. As we add more elements it takes more time to add elements and the trend is linear. But if we started the time before appending more elements, we get an exponential graph as shown in Fig. 3 and Fig. 4.

Finally, dynamic arrays are one of the most fundamental data structures due to their flexibility with varying data sizes. We explored why increasing an array's capacity takes some time, but it allows most of our appends to be almost instantaneous. The same cannot be said about the slow array with an increased capacity of 1 each time. The dynamic array's ability to manage memory through amortized constant-time operations allows it to outperform the slow array significantly, especially as the number of elements grows larger.

We also learned how we can implement our dynamic arrays in code. While Python's list is already a dynamic array, we learn a great deal about how it fundamentally works. And the code does not have to be too complex. It is quite frankly a few lines. If we add more functions to our dynamic array, it can be just like Pythons list. If you check out the full code, you will see that I did add more functions to the dynamic array.

The mathematical underpinning of arrays shows that even if the resizing operations can be costly, the overall time complexity of appending elements remains linear, making dynamic arrays an optimal choice for many programming tasks. If we can understand this behavior, we can design more efficient programs.

In the end, the dynamic array stands as a testament to the power of adaptability—illustrating how, like in biology, survival in programming depends not just on strength, but on the ability to change.