Setting the stage

Our dynamic array initially holds an array of size 0 and capacity 1. It looks like the below.

Index	0
Value	

We want to derive the cost of making n appends to the dynamic array. Each time we append a value we come across one of two possible scenarios:

Scenario 1: The underlying array is not yet full

In this case we haven't reached full capacity yet. Figuratively speaking, there are still seats left on the bus and we can fit more people onboard. This scenario is simple to handle because we simply insert the new value at the logical end of the underlying array. For example, let's append the value 1. It gets placed at index position 0 since that's where the current end of our dynamic array is located.

Index	0
Value	1

All we've done here is write a value – it's a simple one step process.

1. Append the new value to the underlying array (cost = 1 "unit")

Scenario 2: The underlying array is full

In this case we can't fit any more values in the underlying array – our bus is full. At this point the only logical course of action is to throw away the old bus and buy a larger one. Of course, we'll need to move the existing passengers onto the new bus. Afterward, we can let the latest passenger come on board. In other words, there's a three step process when we try to append to a dynamic array whose underlying array has reached capacity.

1. Create a new array that can hold twice as many values as the old array
2. Copy each value from the old array to the new array
3. Append the new value to the new array

We ignore the cost of creating a new array – our analysis will still yield the same big-O result. This takes care of the cost of the first step. We hand-wave over it and forget about it.

1. ~~Create a new array that can hold twice as many values as the old array~~
2. Copy each value from the old array to the new array (cost = (# of values in old array) x 1 unit)
3. Append the new value to the new array (cost = 1 unit)

Now if we append the value 2 to the dynamic array

Index	0
Value	1

Index	0	1
Value	1	2

We create a new array of size 2, copy over any values from the old array (in this case the value 1), then append the new value. Our cost for this particular append is 1 unit (for copying the old value) plus 1 unit (for writing the new value).

Calculating the cost

We can extend our example to include many more appends.

Append #	1	2	3	4	5	6	7	8	9
Write cost	1	1	1	1	1	1	1	1	1
Copy cost	0	1	2	0	4	0	0	0	8
Capacity after append	1	2	4	4	8	8	8	8	16

The key takeaway from the table is that our total cost can be expressed as the sum of all the write costs plus all the copy costs.

$$\begin{aligned} total\ cost &= total\ write\ cost + total\ copy\ cost \\ &= (write\ cost \cdot number\ of\ writes) + total\ copy\ cost \end{aligned}$$

Let's tackle the easy part first. Each time we append a value, a write operation has to occur. Consequently, we incur a cost of 1 unit. If we append n values, we have to make n writes. Then,

$$\begin{aligned} total\ write\ cost &= 1 \cdot n \\ &= n \end{aligned}$$

Now we have to figure out what the total copy cost is when we make n appends.

$$total\ cost = n + total\ copy\ cost$$

But what's the total copy cost?

If we look closely, we can observe a pattern in the copy costs. Each time we resize and copy the old values to a new array, the copy cost doubles.

$$total\ copy\ cost = 1 + 2 + 4 + 8 + \cdots + cost\ of\ the\ last\ resize$$

But what is the cost of the last resize? Let's assume that the number of appends n is some power of 2. This is the same assumption that the exploration makes, so we'll do the same to be consistent. This means that the number of appends is in $\{2, 4, 8, 16, 32, 64, \ldots\}$. According to our table, if n is 4, the last resize has a cost of 2 (at append #3). If n is 8, the last resize has a cost of 4 (at append #5). More generally, on the n^{th} append, the last resize has a cost of $\frac{n}{2}$.

Applying that logic to our formula, we have

$$\textit{total copy cost} = \quad 1+ 2+4+8 + \ldots + \frac{n}{8} + \frac{n}{4} + \frac{n}{2}$$

Figuring out the number of resizes

From here, we can apply two key pieces of information to derive the total copy cost.

1. The total copy cost is the summation of a finite geometric sequence. We know this because on each successive term, the value changes by a common ratio of 2. If you need a quick refresher – see this Khan Academy video
2. The total number of terms (resizes) is $log_2(n)$. How do we know this? The high-level explanation is that we start with 1 and repeatedly multiply by 2 to generate the terms. It takes $log_2(n)$ divisions before we reach n. If you're confused at this point, let's explore what a logarithm represents. It's the exponent y that we need to raise 2 to in order to obtain x.

   $$2^y = x$$

   Taking $\log_2$ on both sides we get, $\log_2 (2^y) = \log_2\ x$

   $$\Rightarrow y\ =\ \log_2 x\ \ [\text{Since}, \log_2 (2^y) = y\ ]$$

   So, we multiply 2 each time starting from 1, we need $log_2(x)$ times to get to x. See why the number of terms from the equation for our total copy cost is $log_2(n)$?

   If you're still not convinced, we can take this one step further and use the formula for the k^{th} term of a geometric sequence.

$$a_k = a_1 r^{k-1}$$

Where a_k is the k^{th} term, a_1 is the first term, and r is the common ratio. What we want to do is solve for k, which is the number of terms in the sequence. We know that the first term is 1 and that the last term is $\frac{n}{2}$. On each successive term we multiply with 2, so the common ratio is 2. Plugging in for the various variables, we have that

$$\frac{n}{2} = 1.\, 2^{k-1}$$

Solving for k,

$\Rightarrow 2(2^{k-1}) = n$

$\Rightarrow 2^k = n$ by the product rule with same base

$\Rightarrow log_2(2^k) = log_2(n)$ taking both sides by log_2

$\Rightarrow (k) log_2(2) = log_2(n)$ by the power rule of logarithms

$\Rightarrow k = log_2(n)$ because $log_b b = 1$

Thus, we have $log_2(n)$ terms. This is the same result as before, when we went for the arguably more intuitive approach. If the logarithm manipulations are confusing, see this page for an excellent explanation of logarithms and derivations for the various rules.

Putting it all together

So now we know that the total copy cost can be expressed by the summation of a finite geometric sequence that has $log_2(n)$ terms and a common ratio of 2. The formula for such a summation is

$$S_k = \frac{a_1(1 - r^k)}{1 - r}, r \neq 1$$

Applying the formula, we get

$$total\ copy\ cost = \frac{(1)\left(1 - 2^{log_2(n)}\right)}{1 - 2}$$

$$= \frac{(1)\left(1 - 2^{log_2(n)}\right)}{(-1)}$$

$$= \frac{\left(1 - n\right)}{(-1)} \quad [\text{Since, } 2^{log_2(n)} = n\,]$$

$$= n - 1$$

So,

$$total\ cost = total\ write\ cost + total\ copy\ cost$$

$$= n + (n - 1)$$

$$= 2n - 1$$

Finally, the amortized big-O runtime complexity

Recall that this is the total cost of n consecutive appends. To get the amortized runtime, we have to spread the total cost over the n appends.

$$amortized\ runtime = \frac{total\ cost}{n}$$

$$= \frac{2n - 1}{n}$$

$$= \frac{2n}{n} - \frac{1}{n}$$

$$= 2 - \frac{1}{n}$$

$$\leq 2 - 0 \quad \text{When } n \text{ is very large } \frac{1}{n} \approx 0$$

$$= 2$$

Thus, the amortized runtime complexity of the above dynamic array insertion is $O(1)$.