# A more efficient algorithm for appending data

James Ko

October 17, 2017

# Contents



**Abstract**

This paper introduces the **growth array**, an array-like data structure designed to be efficient at appending elements. They offer substantial time and memory savings over dynamic arrays when a large number of elements are appended. Growth arrays also support other list operations, such as random access, iteration, and insertion/deletion at an index. However, they perform no better, or slightly worse, than dynamic arrays for those methods. Also, if the size of the data is known beforehand, allocating a raw array with that size and appending to it is faster than appending to a growth array.

# 1 Introduction

In imperative languages, the dynamic array is the most common data structure used by programs[1]. People often want to add multiple items to a list and then iterate it, which dynamic arrays make simple and efficient. But can it be said dynamic arrays have the *most* efficient algorithm for this pattern?

In this paper, I introduce an alternative data structure to the dynamic array, called the **growth array**. It is more efficient than the dynamic array at appending, particularly for large numbers of items. This is due to how it 'grows' once it cannot fit any more items in its buffer.

When dynamic arrays run out of space, they allocate a new buffer, copy the contents of the old one into it, and throw the old one away. However, growth arrays are less wasteful. Instead of throwing away the filled buffer, they keep it a part of the data structure. The new buffer they allocate represents a continuation of the items from the old buffer. For example, if the old buffer contained items $0 - 31$, the new buffer would contain items 32 and beyond. Because of this, growth arrays allocate less memory to store the same number of items, and they do not need to copy items from the old buffer to the new one.

Growth arrays have some caveats, however. They perform no better, or slightly worse, than dynamic arrays with respect to other operations. In particular, random access involves significantly more instructions than it does for dynamic arrays. Additionally, since growth arrays are not contiguous in memory, they may have poorer locality than dynamic arrays, and cannot be passed to external code that accepts contiguous buffers.

It is also worth mentioning that if the size of the data is known in advance, both dynamic and growth arrays are completely unnecessary. A raw array could simply be allocated with the known size, and items could be appended more quickly to it. Thus, growth arrays are only advantageous for scenarios where an unknown amount of data will be appended.

# 2 Fields and Properties

In subsequent sections, I will implement algorithms for both dynamic and growth arrays. In this section, I define the **fields** and **properties** these data structures are assumed to have for those algorithms. **Fields** are variables associated with an object that may be read from or written to. **Properties** are trivial, constant-time methods that do not change state.

[1]

If $L$ is a dynamic array, then it is assumed to have the following fields:

- $L.Buf$ - The **buffer**, or raw array, that $L$ stores its items in.
- $L.Size$ - The number of items in $L$.

As a dynamic array, $L$ is also given the following properties. $:=$ denotes a definition, as opposed to $=$ which checks equivalence. Functions that return boolean values are suffixed with ?.

---

▷ Returns the capacity of $L$
$L.Cap := L.Buf.Len$

▷ Returns whether $L$ is full
$L.Full? := L.Size = L.Cap$

---

If $L$ is a growth array, then it is assumed to have the following fields:

- $L.Head$ - The **head** of $L$. It returns the buffer we are currently adding items to.
- $L.Tail$ - The **tail** of $L$. It returns a dynamic array of buffers that have already been filled. This can be thought of as a two-dimensional array.
- $L.Size$ - The number of items in $L$.
- $L.Cap$ - The **capacity** of $L$. It returns the maximum number of items $L$ can hold without resizing.

As a growth array, $L$ is also given the following properties:

# 3 Common Operations

In this section, I implement common operations for dynamic and growth arrays, and analyze their time complexity. I also analyze space complexity of operations that allocate memory.

In order to highlight the benefit of growth arrays, I must define a new notation for time complexity. The reason for this is, for certain operations, growth arrays are only better than dynamic arrays by a constant factor. For example, dynamic arrays might allocate roughly $2n$ memory for appending $n$ items, while growth arrays would allocate roughly $n$ memory. Even though growth arrays are

▷ Returns whether $L$ is empty
$L.Empty? := L.Size = 0$

▷ Returns whether $L$ is full
$L.Full? := L.Size = L.Cap$

▷ Returns the capacity of $Head$
$L.Hcap := L.Head.Len$

▷ Returns the size of $Head$
▷ **Rationale:** $Cap - Hcap$ is the total capacity of the buffers in $Tail$.
▷ Then, $Size - (Cap - Hcap)$ is the number of items that were added
▷ after depleting the buffers in $Tail$.
$L.Hsize := L.Size - (L.Cap - L.Hcap)$

clearly better in this regard, the big-O space complexity for both data structures would be the same, $O(n)$.

My goal is to be able to compare the coefficients of the highest-order terms in both expressions. For example, I would like to take the ratio $\frac{2n}{n}$, see that it is 2, and conclude that dynamic arrays allocate roughly twice as much as growth arrays for large $n$.

To do this, I define an alternative to big-O, which I will call "big-P notation". I define $P(f(n))$ to be the class of functions $g$ such that $g(n) \to f(n)$ for large $n$. Formally,

$$g \in P(f(n)) \leftrightarrow \lim_{n\to\infty} \frac{g(n)}{f(n)} = 1$$

Like $O$, $P$ can be distributed and "un-distributed" over arithmetic operations. I define

$$\begin{aligned} P(f(n)) + P(g(n)) &:= P(f(n) + g(n)) \\ P(f(n)) - P(g(n)) &:= P(f(n) - g(n)) \\ P(f(n)) \cdot P(g(n)) &:= P(f(n) \cdot g(n)) \\ \frac{P(f(n))}{P(g(n))} &:= P\left(\frac{f(n)}{g(n)}\right) \end{aligned}$$

Notice that while $O(2n) = O(n)$, $P(2n) \neq P(n)$. However, $P$ still retains some nice properties of $O$. In particular, it eliminates lower-order terms from

consideration: $P(n) = P(n + \log_2 n)$, for example. This is stated formally in the following theorem:

**Theorem 3.1.** *If* $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$*, then* $P(f(n) + g(n)) = P(f(n))$.

To prove this, we will need the following lemma.

**Lemma 3.2.** *Each function* $f(n)$ *has a unique class of functions under* $P$*. That is, suppose* $f \in P(g(n))$ *and* $f \in P(h(n))$*. Then* $P(g(n)) = P(h(n))$.

*Proof.* By definition, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$ and $\lim_{n\to\infty} \frac{f(n)}{h(n)} = 1$. Flipping the first equation and multiplying, we receive

$$\begin{aligned} \left(\lim_{n\to\infty} \frac{g(n)}{f(n)}\right)\left(\lim_{n\to\infty} \frac{f(n)}{h(n)}\right) &= 1 \\ \lim_{n\to\infty} \frac{g(n)}{h(n)} &= 1 \\ g &\in P(h(n)) \end{aligned}$$

Now, consider any $g' \in P(g(n))$. Since

$$\begin{aligned} \lim_{n\to\infty} \frac{g'(n)}{g(n)} &= 1 \\ \left(\lim_{n\to\infty} \frac{g'(n)}{g(n)}\right)\left(\lim_{n\to\infty} \frac{g(n)}{h(n)}\right) &= 1 \\ \lim_{n\to\infty} \frac{g'(n)}{h(n)} &= 1 \end{aligned}$$

it follows that $g' \in P(h(n))$, and thus $P(g(n)) \subseteq P(h(n))$. By a similar argument, it can be shown $P(h(n)) \subseteq P(g(n))$. Thus it must be true that $P(g(n)) = P(h(n))$. □

Now 3.1 can be proved as follows:

*Proof.*

$$\begin{aligned}\lim_{n\to\infty}\frac{f(n)+g(n)}{f(n)} &= \lim_{n\to\infty}\frac{f(n)}{f(n)} + \lim_{n\to\infty}\\ &\quad\frac{g(n)}{f(n)}\\ &= 1+0\\ &= 1\end{aligned}$$

Then $f(n)+g(n) \in P(f(n))$. Since it is also true that $f(n)+g(n) \in P(f(n)+g(n))$, it follows from 3.2 that $P(f(n)) = P(f(n)+g(n))$. □

## 3.1 Appending

### Description

Appending is the most common operation on dynamic arrays[2]. Growth arrays improve the performance of appending in two ways: by allocating less memory, and reducing the amount of copying.

#### Dynamic array implementation

I will first implement appending for dynamic arrays. Let $L$ be a dynamic array. The following definitions are used in the code:

**initial capacity** The capacity of a dynamic a array when one item is appended to it. I denote this $C_f$.
(The capacity of any dynamic array with size zero should be zero.)

**growth factor** The factor by which the current capacity is multiplied to get the new capacity when $L$ is resized. I denote this with $G$.

#### Time complexity

Before I analyze the time complexity of [append], I consider a different method for measuring its cost. Suppose I start with an empty list and $n$ elements are appended. How many times is an element stored in an array? I will term the answer to this question the **write cost** of $n$ appends, and denote it $W(n)$.

In the above code, one write is performed after the conditional, so it is apparent that $W(n) \geq n$ after $n$ appends. However, [resize] also does some

[2]

writing, so in order to find a precise formula for $W(n)$, I need to analyze when [resize] is called. Consider the following lemma:

[lemma] Let $L$ by a dynamic array. As $n$ are appended, the sequence of capacities $L$ takes on is

$$S_C = 0,\ C_f,\ GC_f,\ G^2C_f,\ \ldots\ G^{\lceil \log_G n - \log_G C_f \rceil} C_f$$

.

[proof] The capacity starts out at zero and becomes $C_f$ once one element is appended. From then on, it can only grow by a factor of $G$ via [resize], so if $G^iC_f$ is the current capacity then $G^{i+1}C_f$ must be the next capacity. By induction, the rest of the sequence is

$$\left\{G^iC_f\right\}_{i=1}^{k}$$

for some $k$. Since the capacity is only as big as is needs to be, namely greater than or equal to $n$, $k$ is the smallest integer for which $G^kC_f \geq n$. Using this property, I now derive $k$.

$$\begin{aligned} G^kC_f &\geq n \\ G^k &\geq \frac{n}{C_f} \\ k &\geq \log_G n - \log_G C_f \\ k &\geq \log_G n - \log_G C_f > k-1 \\ k &= \lceil \log_G n - \log_G C_f \rceil \end{aligned}$$

Then the final term in the sequence is $G^{\lceil \log_G n - \log_G C_f \rceil} C_f$, completing the proof.

[make lemma] Now, consider that to grow by $G$ [resize] must be called. So $G^kC_f$ being present in $S_C$ means [resize] was called at size $G^{k-1}C_f$ for $k \geq 1$. Then [resize] was called at the sizes

$$S_R = 0,\ C_f,\ GC_f,\ G^2C_f,\ \ldots\ G^{\lceil \log_G n - \log_G C_f \rceil - 1} C_f$$

Each time [resize] is called, $i$ items are copied where $i$ is the current size of the dynamic array. Then the total number of items copied for $n$ appends is

$$0 + C_f + GC_f + \ldots + G^{\lceil \log_G n - \log_G C_f \rceil - 1} C_f = \left( \frac{G^{\lceil \log_G n - \log_G C_f \rceil} - 1}{G-1} \right) C_f$$

Finally, appending the writes made for every item in [append], I finally find an explicit formula for $W(n)$.

[ref1]
$$W(n) = n + \left(\frac{G^{\lceil \log_G n - \log_G C_f \rceil} - 1}{G-1}\right) C_f$$

[ref2] To make this expression easier to work with, I will write it in big-P notation to approximate it for large $n$. First I note that $\log_G k \leq \lceil \log_G k \rceil < (\log_G k) + 1$ for any $k > 0$, so $k \leq G^{\lceil \log_G k \rceil} < Gk$. Then

$$W(n) = n + \left(\frac{G^{\left\lceil \log_G\left(\frac{n}{C_f}\right)\right\rceil} - 1}{G-1}\right) C_f$$
$$n + \left(\frac{\frac{n}{C_f} - 1}{G-1}\right) C_f \leq W(n) < n + \left(\frac{G\frac{n}{C_f} - 1}{G-1}\right) C_f$$
$$n + \left(\frac{n - C_f}{G-1}\right) \leq W(n) < n + \left(\frac{Gn - C_f}{G-1}\right)$$
$$P\left(n + \left(\frac{n - C_f}{G-1}\right)\right) \leq P\left(W(n)\right) < P\left(n + \left(\frac{Gn - C_f}{G-1}\right)\right)$$
$$P\left(\left(\frac{G}{G-1}\right) n\right) \leq P\left(W(n)\right) < P\left(\left(\frac{2G-1}{G-1}\right) n\right)$$

**Space complexity**

I wish to find the space allocated while appending $n$ items to a dynamic array. I denote this quantity $S(n)$, and define it as the total length of all arrays allocated during $n$ [append] calls. I now derive a formula for $S(n)$.

From [lemma] I know the exact capacities $L$ takes on as $n$ items are appended. Dynamic arrays also have the property that $i \in S_C$ iff an array of length $i$ was allocated. Then the total length of arrays allocated is

$$\begin{aligned} S(n) &= \sum_{i \in S_C} i \\ &= 0 + C_f + GC_f + G^2 C_f + \ldots + G^{\lceil \log_G n - \log_G C_f \rceil} C_f \\ &= \left(\frac{G^{\lceil \log_G n - \log_G C_f \rceil + 1} - 1}{G-1}\right) C_f \end{aligned}$$

Using the identity in [ref2] again, I bound its big-P complexity:

$$\left(\frac{G^{\frac{n}{C_f}}-1}{G-1}\right)C_f \leq S(n) < \left(\frac{G^{2\frac{n}{C_f}}-1}{G-1}\right)C_f$$

$$P\left(\left(\frac{G^{\frac{n}{C_f}}-1}{G-1}\right)C_f\right) \leq P\left(S(n)\right) < P\left(\left(\frac{G^{2\frac{n}{C_f}}-1}{G-1}\right)C_f\right)$$

$$P\left(\left(\frac{G}{G-1}\right)n\right) \leq P\left(S(n)\right) < P\left(\left(\frac{G^2}{G-1}\right)n\right)$$

**Growth array implementation**

**Time complexity**

I again start off by finding the write cost when $n$ are appended. I give the write cost function a slightly different name, $W'(n)$, so I can compare it to $W(n)$ later. I also use $G'$ and $C'_f$ instead of $G$ and $C_f$ for the growth factor and initial capacity, respectively.

[That lemma] still holds, since all of the assumptions made there are also true for growth arrays. In particular, although growth arrays resize slightly differently than dynamic arrays, the following claim is still true:

[other lemma] The capacity of a growth array grows by a constant factor $G'$ after the first time it is called.

[proof] I prove that the algorithm ensures this using induction.

For the base case, consider the second time [resize] is called: the current capacity is $C'_f$, and the next (total) capacity should be $G'C'_f$. Then a buffer of size $(G'-1)C'_f$ should be appended, and that is what the algorithm does.

For $i \geq 2$, given the algorithm behaves correctly the $i$th time it is called, I wish to show it behaves correctly the $(i+1)$th time. From the inductive hypothesis, when the algorithm is called the $(i+1)$th time, the current capacity is $G'^iC'_f$ and the head capacity is $(G'^i - G'^{i-1})C'_f$. The next (total) capacity should be $G'^{i+1}C'_f$, and the next head capacity should be $(G'^{i+1} - G'^i)C'_f$, which is $G'$ times the current head capacity. The algorithm determines the capacity of the next buffer in this manner, proving the inductive step and completing the induction.

Now I take a look at [resize]. Unlike dynamic arrays, the number of writes performed does not equal $i$ if the current size is $i$. No items are being copied; the only source of writes is appending a buffer to the tail. The goal is to find the number of writes that causes.

Let $W_R(n)$ denote the total number of writes made by [resize]. From [make

lemma], I know that [resize] is called at the sizes

$$S_R = 0,\ C'_f,\ G'C'_f,\ G'^2C'_f,\ \ldots\ G'^{\lceil \log_{G'} n - \log_{G'} C'_f \rceil - 1}C'_f$$

Let $k$ be the number of times the tail is appended to. Since the tail is a dynamic array, the number of writes it makes is $W(k)$. Since [resize] is called $|S_R|$ times and it appends to the tail each time except for the first, $k = |S_R| - 1 = \lceil \log_{G'} n - \log_{G'} C'_f \rceil$. Then the formula for $W_R(n)$ is

$$W_R(n) = k + \left( \frac{G^{\lceil \log_G k - \log_G C_f \rceil} - 1}{G - 1} \right) C_f$$

where $C_f$ $G$ are the capacity of the *tail* (not the growth array). I finally conclude that

$$W'(n) = n + k + \left( \frac{G^{\lceil \log_G k - \log_G C_f \rceil} - 1}{G - 1} \right) C_f$$

I now approximate this expression with big-P notation. First, I establish an approximation for $k$ using big-O:

$$\begin{aligned} k &= \lceil \log_{G'} n - \log_{G'} C'_f \rceil \\ O(k) &= O(\log n) \end{aligned}$$

Then, it is shown that the $k$-term disappears when analyzing the big-P complexity of $W'(n)$, leaving just $P(n)$:

$$\begin{aligned} P(W'(n)) &= P(n + W(k)) \\ &= P(n) + P(W(k)) \\ &= P(n) + O(\log n) \\ &= P(n) \end{aligned}$$

**Space complexity**

The space function for $n$ [append] calls on a growth array is denoted $S'(n)$. Since growth arrays never throw away item buffers, if the current capacity is $C$ then the total length of item buffers is also $C$. $S'(n)$ is usually greater than $C$,

however; this is because growth arrays not only allocate item buffers, but stores references to them in the tail. Thus the space the tail allocates must also be determined.

I determined in [...] that if $n$ are appended then the tail is appended to $k$ times, where $k = \lceil \log_{G'} n - \log_{G'} C'_f \rceil$. Since the tail is a dynamic array, it allocates $S(k)$ space, where $S$ is the space function of the tail. Through a similar method to [...] it can be shown $O(S(k)) = O(k) = O(\log n)$. Then

$$S'(n) = P(2^{\lceil \log_2 n \rceil}) = O(n)$$

**Time complexity comparison**

**Space complexity comparison**

## 3.2 Indexing

### Description

Indexing is another very common operation on a list. I will call methods that get or set an item at a specified index **get** and **set indexers**, respectively.

## 3.3 Iterating

### Description

**Iteration** of a list is the process of performing some action on each of its elements.

## 3.4 Copying to an array

### Description

Users often want to take list structures, such as dynamic arrays, and convert them into plain arrays. There are multiple reasons why someone would want to do this after they are done appending to the list:

- Plain arrays hold on to exactly the amount of memory needed to hold their elements. However, dynamic and growth arrays allocate more space than necessary to optimize appending new items.

- The user wants to call a function in third-party code that takes a plain array as an argument.
- 
- Plain arrays are contiguous, while growth arrays are fragmented and have worse locality.
- The indexer of growth arrays is several times slower than that of plain arrays, whether the $O(1)$ or $O(\log n)$ implementation is chosen.

# 4 Other Operations

# 5 Implementations

# 6 Benchmarks

# 7 Closing Remarks