A more efficient algorithm for appending data

James Ko

November 5, 2017

Contents

Abstract

This paper introduces **growth arrays**, array-like data structures that are designed for appending elements. When the number of items is not known beforehand but is expected to be large, they are more efficient than dynamic arrays by a constant factor. Growth arrays support all operations dynamic arrays do, such as random access, iteration, and insertion or deletion at an index. However, they perform no better, or slightly worse, than dynamic arrays for operations other than appending.

1 Introduction

In imperative languages, the dynamic array is the most common data structure used by programs. People often want to add multiple items to a collection and then iterate it, which dynamic arrays make simple and efficient. But can it be said they 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 large numbers of items. This is due to how it 'grows' once it cannot fit 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 caveats, however. They perform no better, or slightly worse, than dynamic arrays for operations other than appending. Random access, in particular, involves many more instructions than it does for dynamic arrays. Also, 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 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 to it just as quickly. Thus, growth arrays are only beneficial for cases where the amount of data to be appended is unknown, but is expected to be large.

2 Predefined Functions

I assume the following functions, which I will use in my algorithms, are already defined by the runtime.

▷ Copies *len* items from *source* to *dest*
$Array\ copy(source, dest, len)$

▷ Returns the length of *array*
$array.Len$

▷ Returns a new array with length *len*
$New\ array(len)$

▷ Returns a new, empty dynamic array
$New\ dynamic\ array()$

---

3 Fields and Properties

In subsequent sections, I will implement algorithms for both dynamic and growth arrays. In this section, I define fields and properties for these data structures which the algorithms will use. **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.

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
$L.Full? := L.Size = L.Cap$

When a dynamic array is instantiated, the following code should run:

procedure $Constructor(L)$
    $L.Buf \leftarrow New\ array(0)$
    $L.Size \leftarrow 0$

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 references to buffers that have already been filled. This can be thought of as a two-dimensional array.
  Note: It may seem strange for growth arrays to use dynamic arrays internally. There is little harm from the extra copying and allocations these dynamic arrays perform, however, since only $O(\log n)$ references are appended to them (proved in [ref3]).
- $L.Size$ - 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:

▷ 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 $Tail$.
$L.Hsize := L.Size - (L.Cap - L.Hcap)$

When a growth array is instantiated, the following code should run:

```
procedure Constructor(L)
    L.Head ← New array(0)
    L.Tail ← New dynamic array()
    L.Size ← 0
    L.Cap ← 0
```

4 Asymptotic Notation

4.1 Motivation

In order to highlight the benefit of growth arrays, I will use an alternative 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 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. However, big-O notation does not support this.

4.2 Definition

To analyze time complexity, I will use the $\sim$ relation. It is defined as follows:

$$f \sim g \leftrightarrow \lim_{n \to \infty} \frac{g}{f} = 1$$

This is read as "f is asymptotic to g" or "f and g are asymptotic." **Note:** f and g are used as shorthand to denote $f(n)$ and $g(n)$, respectively.

Notice that while $O(2n) = O(n)$, $2n \not\sim n$. Thus, $\sim$ makes it possible to distinguish between a function that uses n space and one that uses $2n$ space. **Note:** A consequence of this is that bases for logarithms cannot be omitted, like in big-O notation.

4.3 Properties

Here, I define various properties of the $\sim$ relation that will be used in my proofs. Proofs of these properties can be found in the appendix.

The following two theorems state that $\sim$ can "merge", or un-distribute, over arithmetic operations. This is a property shared with big-O.

Theorem 4.1. *P merges over addition and subtraction. That is, for functions f and g,*

$$(\hat{f} \sim f) \wedge (\hat{g} \sim g) \to \begin{cases} (\hat{f} + \hat{g}) \sim (f + g) \\ (\hat{f} - \hat{g}) \sim (f - g) \end{cases}$$

Theorem 4.2. *P merges over multiplication and division. That is, for functions f and g,*

$$(\hat{f} \sim f) \wedge (\hat{g} \sim g) \to \begin{cases} \hat{f}\hat{g} \sim fg \\ \hat{f}/\hat{g} \sim f/g \end{cases}$$

The following theorem states that lower-order terms may be removed. For example, $(n + \log_2 n) \sim n$.

Theorem 4.3. *If $\lim_{n\to\infty} g/f = 0$, then $f + O(g) \sim f$.*

The following theorem shows that $\sim$ is a transitive relation. This property is used implicitly by proofs that chain multiple $\sim$s in the form $f \sim g \sim h$, then conclude that $f \sim h$.

Theorem 4.4. *If $f \sim g$ and $g \sim h$, then $f \sim h$.*

The following theorem states that if two functions are asymptotic, then they belong to the same big-O class.

Theorem 4.5. *If $f \sim g$ and $g = O(h)$, then $f = O(h)$.*

5 Common Operations

In this section, I implement the most common operations for dynamic and growth arrays, then analyze their time complexity. If the operation allocates memory, I also analyze its space complexity.

5.1 Appending

Description

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

Dynamic array implementation

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

initial capacity Denoted by c_0. The capacity of an empty dynamic array.
Assumptions: c_0 is an integer, $c_0 > 0$

growth factor Denoted by g. The factor by which the current capacity is multiplied to get the new capacity when L is non-empty and grows.
Assumptions: $gc_0 \geq c_0 + 1$

```
1: procedure Append(L, item)
2:     if L.Full? then
3:         L.Grow()
4:     L.Head[L.Hsize] ← item
5:     L.Size ← L.Size + 1

6: procedure Grow(L)
7:     new buf ← New array(L.Size × g)
8:     Array copy(L.Buf, new buf, L.Size)
9:     L.Buf ← new buf
```

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 collection 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 code for *Append*, one array store is performed unconditionally, so it is apparent that $w(n) \geq n$ after n appends. However, *Grow* also does some writing, so in order to find a precise formula for $w(n)$, I need to analyze when *Grow* is called. To do this, I use the following lemma:

Lemma 5.1. *Let L by a dynamic array. Let its **capacity sequence**, κ, be the range of values for $L.Cap$ as n items are appended. For $n = 0$, trivially*

$\kappa = (c_0)$. *For* $n > 0$,

$$\kappa = c_0,\ gc_0,\ g^2c_0,\ \ldots\ g^{\max(\lceil \log_g n - \log_g c_0 \rceil, 0)}c_0$$

Proof. I use the following properties of dynamic arrays:

1. The capacity of an empty dynamic array is c_0.
2. The capacity of a dynamic array can only grow by g.
3. The capacity is as small as possible. Put formally, if κ_i is the capacity for n items, then $\kappa_i \geq n$ but $n > \kappa_{i-1}$. (By convention, $\kappa_{-1} = 0$.)

Assumption (1) immediately shows $\kappa_0 = c_0$. Assumption (2) shows that if $g^i c_0$ is the current capacity, then $g^{i+1}c_0$ must be the next capacity. By induction, $\kappa = \left(g^i c_0\right)_{i=0}^{\lambda}$ for some whole number λ.

The final value of the sequence, κ_λ, is the capacity needed for n items. By assumption (3), $\kappa_\lambda \geq n > \kappa_{\lambda-1}$. Consider the case when $n > c_0$: it must be true that $\kappa_\lambda > c_0$, so $\lambda \geq 1$. Since $\lambda - 1 \neq 0$, $\kappa_\lambda = g^\lambda c_0$ and $\kappa_{\lambda-1} = g^{\lambda-1}c_0$. Then

$$\begin{aligned} g^\lambda c_0 &\geq n > g^{\lambda-1}c_0 \\ g^\lambda &\geq \frac{n}{c_0} > g^{\lambda-1} \\ \lambda &\geq \log_g n - \log_g c_0 > \lambda - 1 \end{aligned}$$

Since λ is an integer,

$$\lambda = \lceil \log_g n - \log_g c_0 \rceil$$

Now consider the case when $n \leq c_0$. By assumption (3), $n > \kappa_{\lambda-1}$. $\lambda - 1$ must then equal -1, since any other value would imply $n > \kappa_{\lambda-1} \geq c_0$. Thus $\lambda = 0$.

It was shown $\lambda \geq 1 \geq 0$ for the first case, and it can be shown $\lceil \log_g n - \log_g c_0 \rceil \leq 0$ for the second case. Then, a general formula for λ is as follows:

$$\lambda = \max(\lceil \log_g n - \log_g c_0 \rceil, 0)$$

The final term in the sequence is $g^\lambda c_0 = g^{\max(\lceil \log_g n - \log_g c_0 \rceil, 0)}c_0$, completing the proof. □

[make corollary] If κ_i exists and $i \geq 1$, then clearly *Grow* must have been called when the size was κ_{i-1}. Then the **growth sequence**, the sizes for which *Grow* is called when n items are appended, is κ with the last term removed:

$$\gamma = \kappa \setminus \{\kappa_\lambda\}$$

When *Grow* is called and the current size is γ_i, the algorithm copies κ_i items. Then the total number of items copied when n items are appended is:

$$\begin{aligned}\sum_i \gamma_i &= c_0 + gc_0 + \ldots + g^{\lambda-1}c_0 \\ &= \left(\frac{g^\lambda - 1}{g-1}\right) c_0\end{aligned}$$

Counting the writes made for each item by *Append*, an explicit formula for $w(n)$ is as follows: [ref1]

$$w(n) = n + \left(\frac{g^\lambda - 1}{g-1}\right) c_0$$

Now, my goal is to approximate $w(n)$ with $\sim$. To make is easier to do so, I will asymptotically bound g^λ which depends on n.

Lemma 5.2. *For $n > c_0$, $n/c_0 \leq g^\lambda < gn/c_0$.*

Proof. It was shown in 5.1 that if $n > c_0$, $\lambda = \lceil \log_g n - \log_g c_0 \rceil \geq 1$. Now note that λ may also be written as $\left\lceil \log_g \frac{n}{c_0} \right\rceil$. Then

$$\begin{aligned}\log_g \frac{n}{c_0} \leq \lambda < \log_g \frac{n}{c_0} + 1 \\ \frac{n}{c_0} \leq \lambda < \frac{gn}{c_0}\end{aligned}$$

as desired. □

Now, I proceed to asymptotically bound $w(n)$.

$$\begin{aligned} w(n) = n + \left(\frac{g^\lambda - 1}{g-1}\right) c_0 \\ n + \left(\frac{n/c_0 - 1}{g-1}\right) c_0 \preceq w(n) \prec n + \left(\frac{gn/c_0 - 1}{g-1}\right) c_0 \\ \left(\frac{g}{g-1}\right) n - \left(\frac{c_0}{g-1}\right) \preceq w(n) \prec \left(\frac{2g-1}{g-1}\right) n - \left(\frac{c_0}{g-1}\right) \\ \left(\frac{g}{g-1}\right) n \preceq w(n) \prec \left(\frac{2g-1}{g-1}\right) n \end{aligned}$$

Space complexity

I wish to find the space allocated when n items are appended to a dynamic array. I call this quantity the **space cost**, denote it $s(n)$, and define it as the total length of buffers allocated by n *Append* calls. Now, I derive a formula for $s(n)$.

First, from the definition of *L.Cap*, note that a dynamic array's capacity is the length of the buffer it stores its items in. Then a buffer of length c is allocated at some point if and only if $c \in \kappa$. Then the total length of those buffers is

$$\begin{aligned} s(n) &= \sum_i \kappa_i \\ &= c_0 + gc_0 + g^2c_0 + \ldots + g^\lambda c_0 \\ &= \left(\frac{g^{\lambda+1}-1}{g-1}\right) c_0 \end{aligned}$$

Using Lemma 5.2 again, I asymptotically bound $s(n)$:

$$\begin{gathered} \left(\frac{gn/c_0 - 1}{g-1}\right) c_0 \preceq s(n) \prec \left(\frac{g^2 n/c_0 - 1}{g-1}\right) c_0 \\ \left(\frac{g}{g-1}\right) n - \left(\frac{c_0}{g-1}\right) \preceq s(n) \prec \left(\frac{g^2}{g-1}\right) n - \left(\frac{c_0}{g-1}\right) \\ \left(\frac{g}{g-1}\right) n \preceq s(n) \prec \left(\frac{g^2}{g-1}\right) n \end{gathered}$$

Growth array implementation

Time complexity

I again start off by finding the write cost for n items. 5.1 still holds, since all of the assumptions made there still hold for growth arrays. In particular, although growth arrays use a different growth algorithm than dynamic arrays, the following claim is still true:

Lemma 5.3. *The capacity of a non-empty growth array grows by the constant factor g.*

Proof. I prove that the *Grow* algorithm always enforces this using induction. I induct on the number of times *Grow* is called, k, showing that for all $k \geq 2$, *Grow* behaves correctly when called the kth time. (At $k = 1$, the growth array is empty, so its behavior is not relevant to this lemma.) I will let $c_i(k)$ and $c_f(k)$ denote the initial/final capacities and $h_i(k)$ and $h_f(k)$ denote the initial/final head capacities for the kth call, respectively.

For $k = 2$, $c_i(2) = c_0$. I wish to show that $c_f(2) = gc_0$. This happens if and only if the next buffer has size $\Delta c(2) = (g-1)c_0$, which the algorithm ensures.

For $k > 2$, by induction $c_i(k) = c_f(k-1) = g^{k-2}c_0$ and $h_i(k) = h_f(k-1) = (g^{k-2} - g^{k-3})c_0$. I wish to show $c_f(k) = g^{k-1}c_0$ and $h_f(k) = (g^{k-1} - g^{k-2})c_0$.

Because $k > 2$, the algorithm will calculate $h_f(k)$ as g times $h_i(k)$. Then

$$\begin{aligned} h_f(k) &= g h_i(k) \\ &= g(g^{k-2} - g^{k-3})c_0 \\ &= (g^{k-1} - g^{k-2})c_0 \end{aligned}$$

and

$$\begin{aligned} c_f(k) &= c_i(k) + h_f(k) \\ &= g^{k-2}c_0 + (g^{k-1} - g^{k-2})c_0 \\ &= g^{k-1}c_0 \end{aligned}$$

as desired. □

Now that 5.3 and consequently 5.1 are proved for growth arrays, I examine the write cost of *Grow*. Unlike dynamic arrays, *Grow* does not make s writes if the current size is s. In fact, no items are copied; the only source of writes is when a buffer is appended to the tail, since the tail is a dynamic array.

Let $w_{Grow}(n)$ denote the total number of writes made by *Grow*, and let $\hat{n}$ be the size of the tail. Consider that [make corollary] also holds true for growth arrays, so *Grow* is called $|\gamma|$ times. A buffer is appended to the tail every time except the first, so the size of the tail is

$$\hat{n} = |\gamma| - 1 = |\kappa| - 2 = \lambda$$

Then the formula for $w_{Grow}(n)$ is simply $\hat{w}(\lambda)$, where $\hat{w}$ denotes the write cost function for dynamic arrays. Finally, adding the writes made by *Append*, the formula for $w(n)$ is

$$w(n) = n + \hat{w}(\lambda)$$

Now, I approximate this expression using $\sim$. To do this, I will first need to find the big-O complexity of λ:

$$\begin{aligned} \lambda &= \max(\lceil \log_g n - \log_g c_0 \rceil, 0) \\ \lambda &\sim \max(\lceil \log_g n - \log_g c_0 \rceil, 0) \\ &\sim \lceil \log_g n - \log_g c_0 \rceil \\ &\sim \log_g n - \log_g c_0 \\ &\sim \log_g n \end{aligned}$$

By [lemma], $O(\lambda) = O(\log_g n) = O(\log n)$.

I now approximate this expression using $\sim$. First, I approximate k using big-O:

$$
\begin{aligned}
k &= \lceil \log_g n - \log_g c_0 \rceil \\
O(k) &= O(\log n)
\end{aligned}
$$

Now, the k-term disappears when using $\sim$ to approximate $w(n)$, leaving just n:

$$
\begin{aligned}
w(n) &\sim n + w(k) \\
&= n + O(\log n) \\
&\sim n
\end{aligned}
$$

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 $k = \lceil \log_g n - \log_g c_0 \rceil$. 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

$$
s(n) \sim 2^{\lceil \log_2 n \rceil} = O(n)
$$

Time complexity comparison

Space complexity comparison

5.2 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.

5.3 Iterating

Description

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

5.4 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.

6 Other Operations

7 Implementations

8 Benchmarks

9 Closing Remarks

Appendices

A Proofs of $\sim$ Properties

A.1 Merging over Arithmetic Operations

Proof (Theorem 4.1). □

Proof (Theorem 4.2). This can be easily shown by multiplying the limits corresponding to $\hat{f}$ and $\hat{g}$:

$$\begin{aligned} \lim_{n\to\infty} \frac{\hat{f}}{f} &= 1 \\ \lim_{n\to\infty} \frac{\hat{g}}{g} &= 1 \\ \left(\lim_{n\to\infty} \frac{\hat{f}}{f}\right)\left(\lim_{n\to\infty} \frac{\hat{g}}{g}\right) &= 1 \cdot 1 \\ \lim_{n\to\infty} \frac{\hat{f}\hat{g}}{fg} &= 1 \end{aligned}$$

it follows that $\hat{f}\hat{g} \sim fg$. The statement about division can be proved by flipping the fraction for $\hat{g}$ before multiplying, resulting in

$$\lim_{n\to\infty} \frac{\hat{f}/\hat{g}}{f/g} = 1$$

This shows $\hat{f}/\hat{g} \sim f/g$. □

A.2 Removal of Lower-Order Terms

Proof (Theorem 4.3). Since

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

it follows that $f+g \sim f$. □

A.3 Transitivity

Proof (Theorem 4.4). By definition, $\lim_{n\to\infty} g/f = 1$ and $\lim_{n\to\infty} h/g = 1$. Multiplying the two equations, $\lim_{n\to\infty} h/f = 1$ which implies $f \sim h$. □