A more efficient algorithm for appending data

James Ko

February 5, 2018

Acknowledgments

I would like to thank my research mentor Gary Rubinstein, who provided invaluable feedback on my paper and guided me throughout the research process. I am also grateful to my teacher Joseph Stern, who thoughtfully made time to review the math with me. Additional thanks goes to Peter Brooks and Joel Brewster Lewis for providing additional insights. Last but not least, thank you to Vladlena Powers for inspiring me to pursue scientific research.

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, growth arrays perform better than dynamic arrays by a constant factor. They support all operations dynamic arrays do, such as random access, iteration, and insertion or deletion at an index. For operations other than appending, however, they perform the same as or slightly worse than dynamic arrays.

Growth arrays have the potential to make a major impact on the field of computer science. If dynamic arrays are replaced with growth arrays across various programming languages and frameworks, it is likely that many programs will see significant performance boosts. Also, other array-based data structures such as circular queues and stacks stand to benefit from the efficiency improvements of growth arrays.

1 Introduction

The dynamic array is a very common data structure used by programs written in imperative languages. People often want to append 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 old one's contents into it, then throw the old one away. Growth arrays are less wasteful in this scenario. Instead of discarding the filled buffer, they simply archive it. The new buffer they allocate represents a continuation of the items from the old buffer. For example, if the old buffer contained items $0 - 9$, the new buffer would contain items 10 and beyond. Because of this technique, growth arrays allocate less memory to store the same number of items, and do not need to copy the contents of old buffers.

Growth arrays have caveats, however. Operations other than appending take the same time or longer than for dynamic arrays. In particular, random access is very costly for growth arrays. They are also not contiguous in memory, which gives them poorer locality than dynamic arrays, and prevents them from being 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 unnecessary. One could simply allocate a raw array with the known size, and append items 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 External Definitions

I assume that the following functions are defined by the runtime. Thus, I will use them in my algorithms without providing definitions for them.

Algorithm 1 Runtime-defined functions

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

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

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

2.1 Constants

I will use the following constants in my algorithms. These constants govern how the algorithms will behave. Specific values for them must be supplied by the person using the algorithms.

initial capacity Denoted by c_0; the capacity of an empty dynamic array. c_0 must be a natural number.

growth factor Denoted by g; the constant ratio of the new capacity to the old one when L grows. g must be greater than 1.

3 Fields and Properties

In later sections, I will implement algorithms for both dynamic and growth arrays. In this section, I will define fields and properties for those algorithms to 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.

Let L be a dynamic array. I will give L 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. (**Note:** For pseudocode, I will use := to denote a definition, and = to denote an equivalence check. Functions that return boolean values are suffixed with ?.)

Algorithm 2 Properties (dynamic array)

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

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

The following code should run when L is instantiated:

Algorithm 3 Constructor (dynamic array)

procedure $Constructor(L)$
$L.Buf \leftarrow New\_array(c_0)$
$L.Size \leftarrow 0$

Now, let L be a growth array. I will give it the following fields:

- $L.Head$ – The **head** of L. It returns the buffer new items are appended to.
- $L.Tail$ – The **tail** of L. (**Note:** The tail is a dynamic array.) It returns a dynamic array of pointers to buffers that are already filled with items. The tail can be thought of as a two-dimensional array. (**Remark:** It may seem strange for a growth array to use the very data structure it is replacing. Lemma 5.4 will show, however, that only $O(\log n)$ many pointers are appended to the tail. Thus, the extra copying and allocations the tail performs is minuscule compared to other work done by the growth 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 before it must grow.

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

Algorithm 4 Properties (growth array)

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

▷ Returns the **head capacity**, or the capacity of $Head$
$L.Hcap := L.Head.Len$

▷ Returns the **head size**, or the size of $Head$
▷ **Explanation:** $Cap - Hcap$ is the number of items in $Tail$.
▷ $Size - (Cap - Hcap)$ is the number of items not in $Tail$ (meaning, in $Head$).
$L.Hsize := L.Size - (L.Cap - L.Hcap)$

The following code should run when L is instantiated:

Algorithm 5 Constructor (growth array)

procedure $Constructor(L)$
$L.Head \leftarrow New\_array(c_0)$
$L.Tail \leftarrow New\_dynamic\_array()$
$L.Size \leftarrow 0$
$L.Cap \leftarrow c_0$

4 Asymptotic Notation

4.1 Introduction

Typically, big-O notation is used to analyze the time or space complexity of a function. In order to highlight the benefit of growth arrays, however, I will use the $\sim$ relation to analyze complexity. This is because growth arrays are only better than dynamic ones by a constant factor for certain operations. For example, dynamic arrays might use roughly $2n$ space for appending n items, while growth arrays would use roughly n space. 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 $2n : n$, 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 Definitions

I will mainly use the $\sim$ relation to analyze complexity. It is defined as follows [1]:

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

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 $2n = O(n)$, $2n \not\sim n$. 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.

I will also introduce two cousins of $\sim$, the relations $\prec$ and $\preceq$. The former is normally defined as $f \prec g \iff \lim_{n\to\infty} \frac{f}{g} = 0$. However, for the purpose of this paper their definitions will be:

$$f \prec g \iff \lim_{n\to\infty} \frac{f}{g} < 1$$

$$f \preceq g \iff f \prec g \vee f \sim g$$

I will call these relations **asymptotic inequalities**. The above inequalities are read "f is asymptotically less than g" and "f is asymptotically less than or equal to g," respectively.

I will also use little-o notation in this paper, which is defined as follows [2]:

$$f = o(g) \iff \lim_{n\to\infty} \frac{f}{g} = 0$$

4.3 Properties

I define various properties of the $\sim$ relation here, which proofs in later sections will use. The properties themselves are proved in the appendix.

Note: In the following theorems, variables with the letters f, g, or h denote functions that are positive for sufficiently large n.

The following theorem states that $\sim$ is a valid equivalence relation.

Theorem 4.1. *$\sim$ is reflexive, transitive, and symmetric.*

The following theorem states that lower-order terms may be removed: for example, $(n + \log_2 n) \sim n$. This is a property shared with big-O.

Theorem 4.2. *$f + o(f) \sim f$.*

Corollary 4.2.1. *If f is unbounded, then $f + c \sim f$ for any constant c.*

The following theorem states that $\sim$ respects addition, multiplication, and division. This is also a property shared with big-O.

Theorem 4.3. *If $f \sim f_0$ and $g \sim g_0$, then*

$$f + g \sim f_0 + g_0$$
$$fg \sim f_0 g_0$$
$$\frac{f}{g} \sim \frac{f_0}{g_0}$$

Corollary 4.3.1. *A $\sim$ relation is preserved when both sides are added, multiplied, or divided by the same positive quantity.*

The following theorem states that asymptotic functions belong to the same big-O class.

Theorem 4.4. *If $f \sim g$ and $g = O(h)$, then $f = O(h)$.*

The following theorem states that asymptotic functions can be interchanged in an asymptotic inequality.

Theorem 4.5. *If $f \sim f_0$ and $f_0 \prec f_1$, then $f \prec f_1$. If $g \sim g_1$ and $g_0 \prec g_1$, then $g_0 \prec g$.*
These statements also hold for $\preceq$.

The following theorem states that asymptotic inequalities can be algebraically manipulated, in the same vein as Corollary 4.3.1.

Theorem 4.6. *A $\prec$ inequality is preserved when both sides are multiplied or divided by the same positive quantity.*
This statement also holds for $\preceq$.

Corollary 4.6.1. *If $f \prec g$, then $\frac{1}{g} \prec \frac{1}{f}$.*
This statement also holds for $\preceq$.

Notice that the above theorem does not include addition. Asymptotic inequalities do not always hold if the same quantity is added to both sides; for example, $1 \prec 2$, but $n + 1 \sim n + 2$. In order for them to hold, certain conditions must be met.

Theorem 4.7. *Suppose $f \prec g$. If $\lim_{n\to\infty} \frac{f}{h} > 0$ and $g = O(h)$, then $f + h \prec g + h$.*
This statement also holds for $\preceq$.

5 Operations

In this section, I implement select operations for dynamic and growth arrays, and analyze their time complexity. I also analyze the space complexity of appending since it allocates memory.

5.1 Appending

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 constants are used by the algorithm:

Algorithm 6 Appending (dynamic array)

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

6: procedure Grow(L)
7:     new_buf ← New_array(g × L.Size)
8:     Array_copy(L.Buf, new_buf, L.Size)
9:     L.Buf ← new_buf
```

Time complexity

It is well-known that appending one item to a dynamic array takes amortized $O(1)$ time [3]. Thus, appending n items takes $O(n)$ time. It will be shown later, however, that growth arrays also take $O(n)$ time for this task, so this does not let us compare time complexities.

I wish to find the ratio $\frac{t_D(n)}{t_G(n)}$, where $t_D(n)$ and $t_G(n)$ denote the average time needed to append n items for dynamic and growth arrays, respectively. This represents how much faster growth arrays are at appending than dynamic ones. I will approximate this ratio using the following question: Suppose n elements are appended to an empty collection. 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 $w(n)$.

I will assume that $t(n) \propto w(n)$, or that the average time for n appends is proportional to their write cost (although this is an oversimplification). Then $\frac{t_D(n)}{t_G(n)} = \frac{w_D(n)}{w_G(n)}$, where w_D and w_G are respectively the write cost functions for dynamic and growth arrays. In this section, I will derive the formula for $w_D(n)$. Because this section concerns dynamic arrays only, I will 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 via *Array_copy*, 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 be 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 to store 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 $\lambda - 1 \geq 0$, $\kappa_\lambda = g^\lambda c_0$ 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 that $\lambda \geq 1 \geq 0$ for the first case, and it can be shown that $\lceil \log_g n - \log_g c_0 \rceil \leq 0$ for the second case. Therefore, 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. □

Corollary 5.1.1. *Let the* ***growth sequence****, γ, of L be the sizes at which Grow is called when n items are appended to L. Then $\gamma = \kappa \setminus \{\kappa_\lambda\}$.*

Proof. If κ_i exists and $i \geq 1$, then clearly *Grow* must have been called when the size was κ_{i-1}, so $\kappa_{i-1} \in \gamma$. Then γ contains every term in κ except for the last, κ_λ, as the corollary states. □

When *Grow* is called and the current size is γ_i, the algorithm copies γ_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 per item by *Append*, an explicit formula for $w(n)$ is as follows:

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

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

Lemma 5.2. *$\frac{n}{c_0} \preceq g^\lambda \prec \frac{gn}{c_0}$. That is, $\frac{n}{c_0} \leq g^\lambda < \frac{gn}{c_0}$ for sufficiently large n.*

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

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

as desired. □

I can build on this inequality to receive asymptotic bounds for $w(n)$:

$$\begin{aligned}\frac{n}{c_0} \preceq g^\lambda \prec \frac{gn}{c_0} & \\ \frac{n}{c_0} \preceq g^\lambda - 1 \prec \frac{gn}{c_0} & \quad \text{(Corollary 4.2.1, Theorem 4.5)} \\ \frac{n}{g-1} \preceq \left(\frac{g^\lambda - 1}{g-1}\right) c_0 \prec \frac{gn}{g-1} & \quad \text{(Theorem 4.6)} \\ \left(\frac{g}{g-1}\right) n \preceq w(n) \prec \left(\frac{2g-1}{g-1}\right) n & \quad \text{(Theorem 4.7)}\end{aligned}$$

This means that, for large n, the number of items written to append n items is roughly between the two bounds shown. For example, suppose $c_0 = 4$, $g = 2$, and 32 items are appended. *Grow* is called at the sizes 4, 8, and 16, so $4 + 8 + 16 = 28$ items are copied. Counting the 32 writes made by *Append*, one per item, the write cost is $w(32) = 28 + 32 = 60$. This is approximately equal to the lower bound, $\frac{2}{2-1} \cdot 32 = 64$.

Now, suppose that one more item is added so that $n = 33$. *Grow* is called again, so the number of items copied becomes $4 + 8 + 16 + 32 = 60$. The write cost becomes $w(33) = 33 + 60 = 93$. This is approximately equal to the upper bound, $\frac{2 \cdot 2 - 1}{2-1} \cdot 33 = 99$.

Intuitively, it makes sense that $w(n)$ should jump from the lower bound to the upper bound when $n = \gamma_i + 1$, since at that point $O(n)$ many items are copied from the old to the new buffer. (In the previous example, $\gamma_i + 1$ is 17.) As new items are added, however, both n and $w(n)$ only increase by 1, and the ratio $\frac{w(n)}{n}$ slowly dwindles toward the lower bound until n reaches $\gamma_{i+1} + 1$.

Space complexity

I wish to find the space allocated when n items are appended to a collection. I call this quantity the **space cost**, denote it $s(n)$, and define it as the total length of buffers allocated by n appends. 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

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

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

$$\begin{aligned} \frac{n}{c_0} &\preceq g^\lambda \prec \frac{gn}{c_0} \\ \frac{gn}{c_0} &\preceq g^{\lambda+1} \prec \frac{g^2 n}{c_0} && \text{(Theorem 4.6)} \\ \frac{gn}{c_0} &\preceq g^{\lambda+1} - 1 \prec \frac{g^2 n}{c_0} && \text{(Corollary 4.2.1, Theorem 4.5)} \\ \left(\frac{g}{g-1}\right) n \preceq s(n) &\prec \left(\frac{g^2}{g-1}\right) n && \text{(Theorem 4.6)} \end{aligned}$$

Using the same intuition from the time complexity section, $s(n)$ approaches the lower bound as $n \to \gamma_i$, and jumps to the upper bound when $n = \gamma_i + 1$. If 100 items were appended to a dynamic array with $c_0 = 11$ and $g = 3$, for example, the wasted space $s(100) - 100$ would be approximately $\frac{3^2}{3-1} \cdot 100 - 100 = 350$. If n were 99, however, it would be approximately $\frac{3}{3-1} \cdot 99 - 99 = 49.5$.

Growth array implementation

The following diagrams should provide intuition on how growth arrays work.

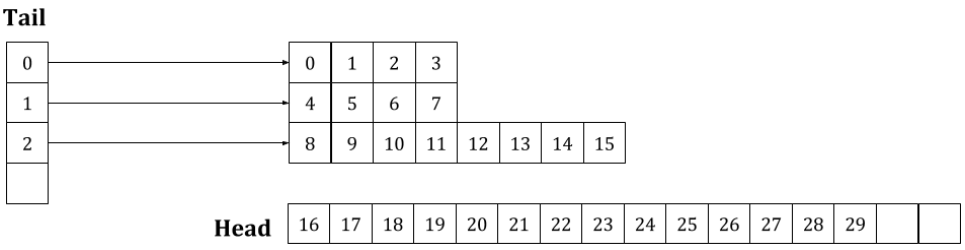

Figure 1: Appending 30 items to a growth array with $c_0 = 4$, $g = 2$

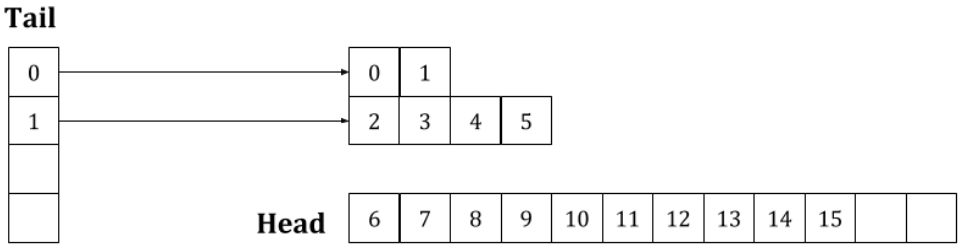

Figure 2: Appending 16 items to a growth array with $c_0 = 2$, $g = 3$

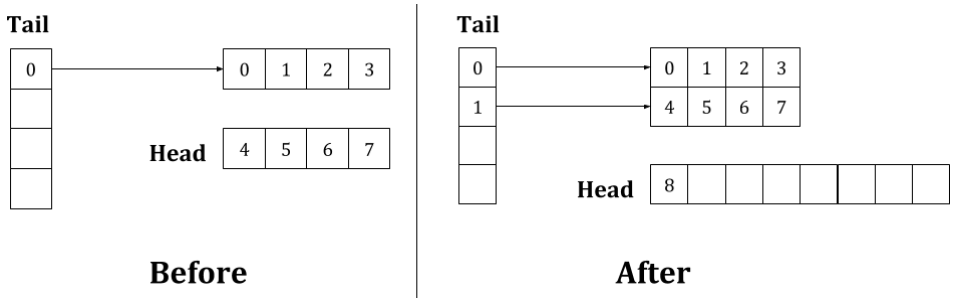

Figure 3: Appending 1 item to a growth array with $c_0 = 4, g = 2$. Initial size: 8

In this section, L denotes a growth array. The following algorithm implements appending for growth arrays.

Algorithm 7 Appending (growth array)

```
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:      L.Tail.Append(L.Head)
8:      if L.Cap = c_0 then
9:          new_hcap ← (g − 1) × c_0
10:     else
11:         new_hcap ← g × L.Hcap
12:     L.Head ← New_array(new_hcap)
13:     L.Cap ← L.Cap + new_hcap
```

Time complexity

It takes $O(n)$ time to append n items to a growth array. This can be easily seen by noting that every statement in the algorithm has amortized $O(1)$ time complexity (ignoring the call to New_array, since memory allocation is non-deterministic).

Now, I focus on finding a formula for growth arrays' write cost function, $w_G(n)$, which will be written as $w(n)$ in this section. I claim that Lemma 5.1 also holds for growth arrays, since they satisfy all properties used by that proof. In particular, although growth arrays use a different growth algorithm than dynamic arrays, they still have the following property:

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

Proof. I prove that the *Grow* algorithm enforces this using induction. I induct on the number of times *Grow* is called, k, showing that for all natural numbers k, *Grow* behaves correctly when called the kth time. I will let c_i and c_f denote the initial/final capacities and h_i and h_f denote the initial/final head capacities for the kth call, respectively.

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

For $k > 1$, by induction $c_i =$ previous $c_f = g^{k-1}c_0$, and $h_i =$ previous $h_f = (g^{k-1} - g^{k-2})c_0$. I wish to show $c_f = g^k c_0$ and $h_f = (g^k - g^{k-1})c_0$. Because $k > 1$, the algorithm will calculate h_f as g times h_i. Then

$$h_f = gh_i = g(g^{k-1} - g^{k-2})c_0 = (g^k - g^{k-1})c_0$$

and

$$c_f = c_i + h_f = g^{k-1}c_0 + (g^k - g^{k-1})c_0 = g^k c_0$$

as desired. □

Since Lemma 5.3 has been proven, Lemma 5.1 and all results based on it must also hold true for growth arrays. Now, I am ready to find the write cost of *Grow*. Unlike dynamic arrays, *Grow* does not make γ_i writes when the current size is γ_i. In fact, *Grow* does not copy *any* items supplied by the user. Writes are only made when a buffer pointer is appended to the tail, since the tail is a dynamic array.

Let $w_\gamma(n)$ denote the total number of writes made by *Grow*, and let n_τ be the size of the tail. Since Corollary 5.1.1 also holds for growth arrays, *Grow* is called $|\gamma|$ times. A buffer is appended to the tail each time *Grow* is called. Thus, the tail's size is

$$n_\tau = |\gamma| = |\kappa| - 1 = \lambda$$

Then the formula for $w_\gamma(n)$ is simply $w_D(\lambda)$, since λ pointers are appended to the tail, which is a dynamic array. Finally, adding the writes made by *Append*, the formula for $w(n)$ is

$$w(n) = n + w_D(\lambda)$$

Now, I approximate $w(n)$ using $\sim$. To do this, I will derive the big-O complexity of λ.

Lemma 5.4. $\lambda = O(\log n)$.

Proof. From Lemma 5.1, $\lambda = \max(\lceil \log_g n - \log_g c_0 \rceil, 0)$. For sufficiently large n, $\lambda = \lceil \log_g n - \log_g c_0 \rceil$. Using $\sim$ to analyze λ, I receive

$$\lambda \sim \lceil \log_g n - \log_g c_0 \rceil \sim \left(\log_g n - \log_g c_0\right) \sim \log_g n$$

(It is trivial to show that $\lceil f(n) \rceil \sim f(n)$ if f is unbounded, since $\lceil f(n) \rceil - f(n)$ is bounded by 1.)

By Theorem 4.4, $O(\lambda) = O(\log_g n) = O(\log n)$ as desired. □

Now when $w(n)$ is approximated with $\sim$, $w_D(\lambda)$ disappears:

$$w(n) = n + w_D(\lambda) = n + O(\lambda) = n + O(\log n) \sim n$$

(The last step is justified by Theorem 4.2.)

This says that when n items are added, the number of items written is approximately n. This is a significant improvement over the asymptotic bounds for $w_D(n)$.

Space complexity

Unlike dynamic arrays, growth arrays do not throw away buffers. This means that if the current capacity is c, then the total length of buffers allocated to store items is also c. Typically, however, $s(n) > c$. This is because the tail of growth arrays (which is a dynamic array) also allocates buffers to store buffer pointers. The space the tail allocates must be considered in the formula for $s(n)$.

In the previous section, I established that $n_\tau = \lambda$. Since κ_λ is the capacity needed to hold n items, and $s_D(\lambda)$ is the space the tail needs to store λ pointers, I conclude that the formula for $s(n)$ is

$$s(n) = \kappa_\lambda + s_D(\lambda)$$

(s_D denotes the space cost function for dynamic arrays.)

Now, I wish to approximate this using $\sim$. Since $\kappa_\lambda = g^\lambda c_0$, it follows from Lemma 5.2 that $n \preceq \kappa_\lambda \prec gn$. From Corollary 4.6.1, $\frac{1}{gn} \prec \frac{1}{\kappa_\lambda} \preceq \frac{1}{n}$. From Theorem 4.6, $\frac{s_D(\lambda)}{gn} \prec \frac{s_D(\lambda)}{\kappa_\lambda} \preceq \frac{s_D(\lambda)}{n}$.

Since $s_D(\lambda) = O(\lambda) = O(\log n)$, the limits of the lower and upper bounds are both 0. From the squeeze theorem, it follows that $\lim_{n\to\infty} \frac{s_D(\lambda)}{\kappa_\lambda} = 0$. Thus, $s_D(\lambda) = o(\kappa_\lambda)$. From Theorem 4.2, one can conclude

$$s(n) = \kappa_\lambda + s_D(\lambda) \sim \kappa_\lambda$$

Using Theorem 4.5, I may substitute $s(n)$ for κ_λ in the first inequality to receive

$$n \preceq s(n) \prec gn$$

This result is significant since it has smaller lower and upper bounds than $s_D(n)$. Since the same values of n as for $s_D(n)$ correspond to when it is close to the bounds for $s(n)$ (namely, when n is approximately γ_i), $s(n)$ is asymptotically less than $s_D(n)$ even though their intervals overlap. This will be proven rigorously in the space complexity comparison section.

Time complexity comparison

In this section, w_D and w_G will denote the write cost functions for dynamic and growth arrays, respectively. My goal is to approximate the ratio $\frac{t_D(n)}{t_G(n)}$ for large n. From my assumption near the beginning of

Section 5.1, this is equal to $\frac{w_D(n)}{w_G(n)}$. I derived earlier that

$$\left(\frac{g}{g-1}\right) n \preceq w_D(n) \prec \left(\frac{2g-1}{g-1}\right) n$$

From Theorem 4.6, I may divide all sides of this inequality by n to receive

$$\frac{g}{g-1} \preceq \frac{w_D(n)}{n} \prec \frac{2g-1}{g-1}$$

It was shown earlier that $w_G(n) \sim n$. From Theorem 4.3, $\frac{w_D(n)}{n} \sim \frac{w_D(n)}{w_G(n)}$. From Theorem 4.5, it follows that

$$\frac{g}{g-1} \preceq \frac{w_D(n)}{w_G(n)} \prec \frac{2g-1}{g-1}$$

For $g = 2$, this can be interpreted as "For large n, dynamic arrays make approximately 2 to 3 times as many writes as growth arrays. Thus, for large n, they are about 2 to 3 times slower than growth arrays."

Space complexity comparison

Let $s_D(n)$ and $s_G(n)$ be the space cost functions for dynamic and growth arrays, respectively. I wish to approximate the ratio $\frac{s_D(n)}{s_G(n)}$ for large n. Recall that

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

and that $s_G(n) \sim \kappa_\lambda = g^\lambda c_0$. Dividing $s_D(n)$ by $s_G(n)$, I receive

$$\begin{aligned}
\frac{s_D(n)}{s_G(n)} &\sim \frac{1}{g^\lambda c_0} \cdot \left(\frac{g^{\lambda+1}-1}{g-1}\right) c_0 && \text{(Theorem 4.3)} \\
&= \frac{g - 1/g^\lambda}{g-1} \\
&\sim \frac{g}{g-1} && \text{(Theorem 4.2)}
\end{aligned}$$

For $g = 2$, this can be interpreted as "For large n, dynamic arrays use approximately twice as much space as growth arrays."

5.2 Indexing

Indexing, or accessing the ith element of a collection, is another common dynamic array operation. Indexing a growth array involves more instructions than indexing a dynamic array. Depending on how it is implemented, indexing for growth arrays runs in either constant or logarithmic time.

Dynamic array implementation

The algorithm for indexing a dynamic array is trivial, and will not be shown here. It simply reads or writes to the ith element of $L.Buf$ (where i is zero-based). It is well-known that this algorithm runs in $O(1)$ time. It also has the benefit of using very few instructions, which will not be the case for growth arrays' algorithm.

Growth array implementation

Indexing a growth array is slower and more complex than indexing a dynamic array. However, this does not mean that accessing data of a growth array is necessarily slower than accessing that of a dynamic array. Refer to Sections 5.3 and 5.4 for methods to quickly access growth arrays' data.

Although growth arrays have poor random access performance, I include this section for two reasons. 1) Dynamic arrays are indexed frequently. In order for growth arrays to be a viable replacement for them, it should be possible to index growth arrays too. 2) I wish to show that it is possible to index growth arrays in constant time.

$O(\log n)$ **implementation**

The first algorithm I will demonstrate is a naïve implementation that runs in logarithmic time.

Note: My algorithms assume all arguments passed to them are valid. Thus, arguments are never checked.

Algorithm 8 Random access (growth array), logarithmic time

```
function Get_item(L, index)
    i ← index
    for buf in L.Tail do
        if i < buf.Len then
            return buf[i]
        i ← i − buf.Len
    return L.Head[i]
```

I know that this algorithm runs in $O(\log n)$ time since it was shown earlier that the size of the tail, n_τ, equals λ, and Lemma 5.4 states that $\lambda = O(\log n)$. Aside from iteration over the tail, all statements run in constant time, so the time complexity is precisely $O(\log n)$.

I present this algorithm alongside the constant time one since it is easier to understand, and the latter is often slower if special circumstances are not met.

$O(1)$ **implementation**

In this section, I will present a constant time algorithm for indexing a growth array. Before I do so, however, I must establish a mathematical justification for it.

Suppose I want to get the ith element of a growth array, where i is zero-based. I assume that i is a valid index, or that $0 \leq i < L.Size$. In order to locate the desired element, I must find two things: the buffer that holds the item, and the index of the item within that buffer. For example, referring to Figure 5.1: if i was 10, I would want the item at index 2 of the tail's third buffer. If i was 20, I would want the item at index 4 of the head.

Now, consider that every buffer except the head is stored inside the tail. Thus, such buffers can be uniquely identified by their index in the tail. I will refer to this quantity as the **buffer index** and denote it i_B. I wish to assign the head a buffer index as well, so that each buffer has a unique ID. Since the head follows the tail's last buffer, which has $i_B = n_\tau - 1$, I will let the head's i_B be n_τ.

I define a helper function, Get_buf, that gets the buffer associated with a given i_B. i_B is assumed to be valid; that is, $0 \leq i_B \leq n_\tau$.

Algorithm 9 Helper function

```
function Get_buf(L, i_B)
    if i_B < L.Tail.Size then
        return L.Tail[i_B]
    else
        return L.Head
```

I will call the index of the desired element within the buffer the **element index**, and denote it i_E.

If I define a helper function *Decompose* that splits i into the ordered pair $(i_B,\ i_E)$, then *Get_item* may be written as follows:

Algorithm 10 Random access (growth array), constant time

```
function Get_item(L, index)
    (i_B, i_E) ← Decompose(index)
    return L.Get_buf(i_B)[i_E]
```

Now, the task is to find formulae for i_B and i_E in terms of i, in order to implement *Decompose*. Referring to Figure 5.1: *Decompose*(10) should return $(2,\ 2)$. *Decompose*(20) should return $(3,\ 4)$.

Lemma 5.5. *The formulae for i_B and i_E are*

$$i_B = \lambda|_{n=i+1}$$
$$i_E = i - \gamma_{i_B-1}$$

(By convention, $\gamma_{-1} = 0$.)

Proof. Appending new items does not change the index of an item that is already in the list. Thus, finding the element at index i is equivalent to finding the last element when $n = i + 1$.

The last element always resides in the head, so $i_B = n_\tau|_{n=i+1}$. As shown earlier, $n_\tau = \lambda$, so $i_B = \lambda|_{n=i+1}$.

To determine i_E, consider the identity:

$$n = \#\text{ of items in tail} + \#\text{ of items in head}$$

(**Note:** The first quantity on the right-hand side is not n_τ. n_τ is the number of buffer pointers in the tail.)

First, I will determine how many items the tail holds. In the case where $i_B = 0$, then because $n_\tau = i_B$, the tail contains 0 buffer pointers and thus 0 items (which equals γ_{-1}). If $i_B > 0$, the number of items in the tail is the last size at which *Grow* was called, or the last term of γ. This quantity is $\gamma_{\lambda-1}$, or γ_{i_B-1}.

The desired item is the growth array's last element, which implies it is also the head's last element. Thus i_E is the head's last valid index, so the head's size is i_E+1. Finally, from the premise, $n = i+1$. Substituting all values into the above identity, I receive

$$i + 1 = \gamma_{i_B-1} + (i_E + 1)$$
$$i_E = i - \gamma_{i_B-1}$$

completing the proof. □

Using the formulae $\lambda = \max(\lceil \log_g n - \log_g c_0 \rceil, 0)$ and $\gamma_k = g^k c_0$ for $k \geq 0$, I now implement the *Decompose* function:

Algorithm 11 Helper function

```
function Decompose(index)
    i_B ← max(⌈log_g(index + 1) − log_g c_0⌉, 0)
    if i_B > 0 then
        i_E ← index − g^(i_B − 1) × c_0
    else
        i_E ← index
    return (i_B, i_E)
```

At first glance, *Decompose* appears to be expensive: logarithms and exponentiation normally imply use of floating-point instructions, which tend to be slower than integer-based ones [4]. However, in the special case where $g = 2$ and $c_0 = 2^\varepsilon$ for some constant whole number ε, I claim that i_B and i_E can be computed without use of floating-point instructions.

To see this, note that i_B becomes $\max(\lceil \log_2(i+1) \rceil - \varepsilon, 0)$. Using the concept of de Bruijn sequences, one can index the leftmost 1 in the binary representation of i in constant time [5]. (For example, if $i = 9 = 1001$ in binary, the index of the leftmost 1 starting from the right is 3.) This index is the same as $\lfloor \log_2 i \rfloor$. Using the fact that $\lceil \log_2(i+1) \rceil = \lfloor \log_2 i \rfloor + 1$ (as there cannot be natural numbers between $\log_2 i$ and $\log_2(i+1)$), one can easily find i_B. Finally, calculating $g^{i_B - 1}$ to find i_E becomes a simple bit shift when $g = 2$.

5.3 Iterating

Informally, to **iterate** a collection is to loop over each of its elements. The syntax "**for** *item* **in** *collection*" implicitly uses the iteration algorithm of *collection*.

It is very common to iterate a dynamic array once items are appended to it. In this section, I will show that growth arrays perform comparably to dynamic arrays for iteration.

Dynamic array implementation

The algorithm for iterating dynamic arrays is well-known: let i range from 0 to $n - 1$ and, for each i, do something with $L[i]$. Ignoring what is done with each item, the time complexity of this operation is $O(n)$.

Growth array implementation

The iteration algorithm for growth arrays is not quite as simple. As shown in Section 5.2, growth arrays' random access algorithm can be expensive; thus, it is best avoided. The following algorithm implements iteration without using the random access algorithm.

Algorithm 12 "**for** *item* **in** L" algorithm (growth array)

```
for buf in L.Tail do
    for item in buf do
        do something with item
for i = 0, i < L.Hsize, i ← i + 1 do
    do something with L.Head[i]
```

Since this algorithm loops over n items, its time complexity is $O(n)$. There is nothing to suggest that this algorithm should be substantially slower than dynamic arrays' algorithm. Even though the inner loop is occasionally 'interrupted' by having to switch buffers, the number of such interruptions is $O(\log n)$, so their cost is minuscule. Thus, the only substantial work done by this algorithm is iterating the buffers, which is roughly the same as the work done by dynamic arrays' algorithm. Therefore, both algorithms should run in approximately the same time.

5.4 Converting to a raw array

Users often want to convert dynamic arrays to raw arrays. There are multiple reasons for this: 1) Raw arrays hold on to exactly n memory to store n elements. However, dynamic and growth arrays typically use more than n memory, so that they need not grow every time an item is appended. 2) Often, functions in third-party code only accept raw arrays.

For growth arrays, the case is even more compelling: 3) Since they are not contiguous as a whole, they have worse data locality (although the number of discontinuities is only $O(\log n)$). 4) Raw arrays can be indexed much faster than growth arrays.

In this section, I will show that dynamic arrays and growth arrays have similar performance for this operation.

Dynamic array implementation

It is straightforward to convert a dynamic array to a raw array:

Algorithm 13 Converting to a raw array (dynamic array)

```
function To_raw_array(L)
    raw_array ← New_array(L.Size)
    Array_copy(L.Buf, raw_array, L.Size)
    return raw_array
```

Since n elements must be copied, the time complexity of this function is $O(n)$.

Growth array implementation

The following algorithm converts a growth array to a raw array:

Algorithm 14 Converting to a raw array (growth array)

```
function To_raw_array(L)
    raw_array ← New_array(L.Size)
    for buf in L.Tail do
        Array_copy(buf, raw_array, buf.Len)
    Array_copy(L.Head, raw_array, L.Hsize)
    return raw_array
```

This algorithm copies n items, so its time complexity is $O(n)$. Using the same argument as in Section 5.3, the cost of 'interruptions' from switching buffers is minuscule. Ignoring New_array since it is non-deterministic, the only substantial work done by this function is the multiple $Array\_copy$ calls. They constitute roughly the same work done by dynamic arrays' single $Array\_copy$ call. Thus, To_raw_array should take roughly as long for growth arrays as it does for dynamic arrays.

5.5 Other operations

Growth arrays support other dynamic array operations, such as insertion or deletion at an index, binary search, sorting, etc. I will not give algorithms for these operations in this paper, since they are not as common as the four detailed above. However, I will briefly describe how insertion and deletion can be implemented.

Suppose the user wants to insert an item at index i. If $i = n$, then simply append the item. If $0 \leq i < n$: capture $L[n-1]$ in a local variable. Shift all elements with index $\geq i$ forward, such that their index increases by 1. Append the original value of $L[n-1]$. Finally, store the provided item at index i.

Suppose the user wants to delete the item at index i. Shift all elements with index $> i$ backwards, such that their index decreases by one. (This will overwrite the item at index i.) Decrement $L.Size$ and $L.Hsize$. If this causes $L.Hsize$ to become 0 and currently $L.Tail.Size > 0$, then discard the head and replace it with the last buffer pointed to by the tail. Remove this buffer from the tail.

6 Data Collection and Analysis

To test my claims, I measured the performance of the above operations in a real programming language and runtime. My computing environment consisted of a Windows 10 desktop with an Intel Core i5-4460T 1.90GHz Haswell processor for the appending benchmarks, and a Windows 10 laptop with an Intel Core i5-7200U 2.50GHz Kaby Lake processor for the other benchmarks. On both computers, I used C# 7.0 with the .NET Core 2.1.4 runtime. I used the excellent BenchmarkDotNet library to obtain performance metrics and plotted them with the R package ggplot2.

I compared the performance of a growth array implementation I had written in C# to that of a heavily-optimized dynamic array that was part of the C# standard library, `System.Collections.Generic.List`. The `List` type had parameters $c_0 = 4$ and $g = 2$, while my growth array implementation had $c_0 = 8$ and $g = 2$. (The different values of c_0 were not significant since $\frac{8}{4}$ was a multiple of g, causing the two collections to exhibit similar trends for large n.) I used 57 unique values of n for the appending benchmarks, and 15

different values of n for the other benchmarks. In both cases, input values included increasing powers of 2 up to 32768. For the appending benchmarks, I also included numbers that were one more than a power of 2, numbers that were the average of two consecutive powers of 2, and all numbers less than 10 in my input data. Here are my results:

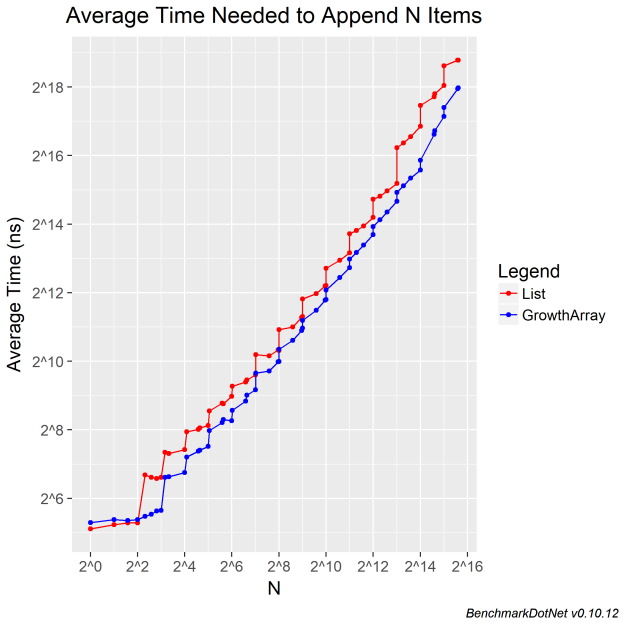

Figure 4: Time measurements for appending

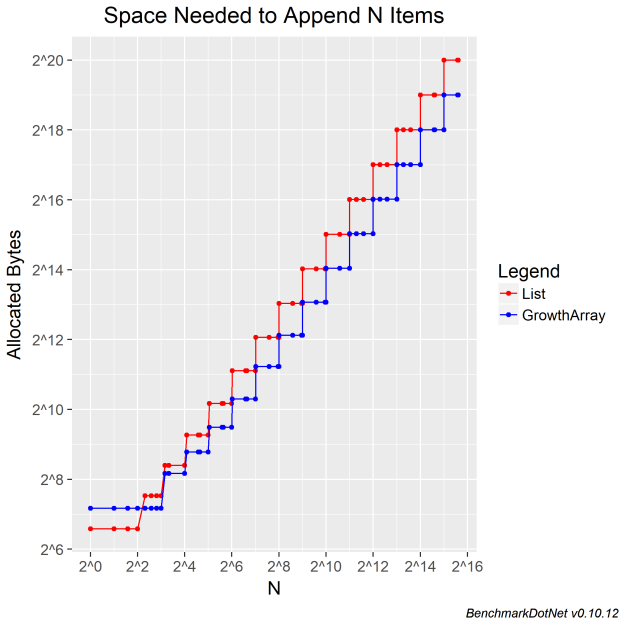

Figure 5: Space measurements for appending

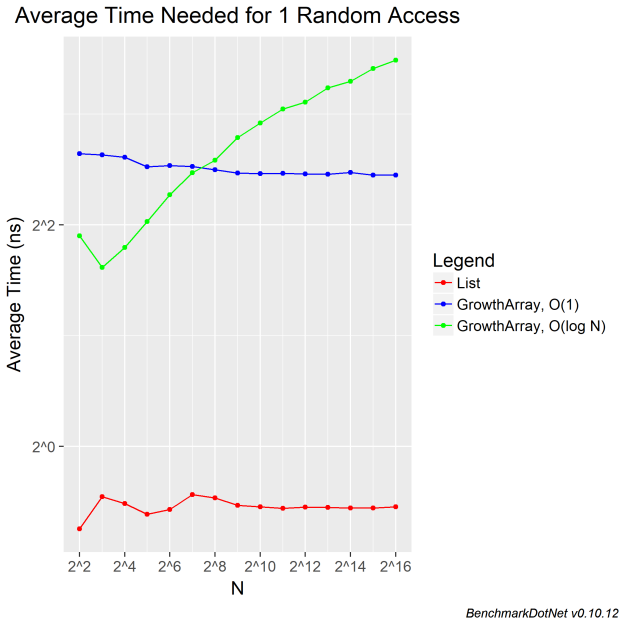

Figure 6: Time measurements for indexing

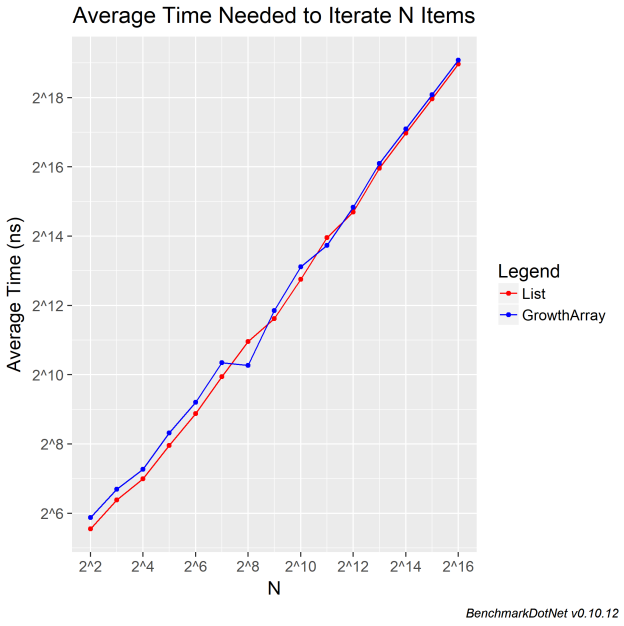

Figure 7: Time measurements for iterating

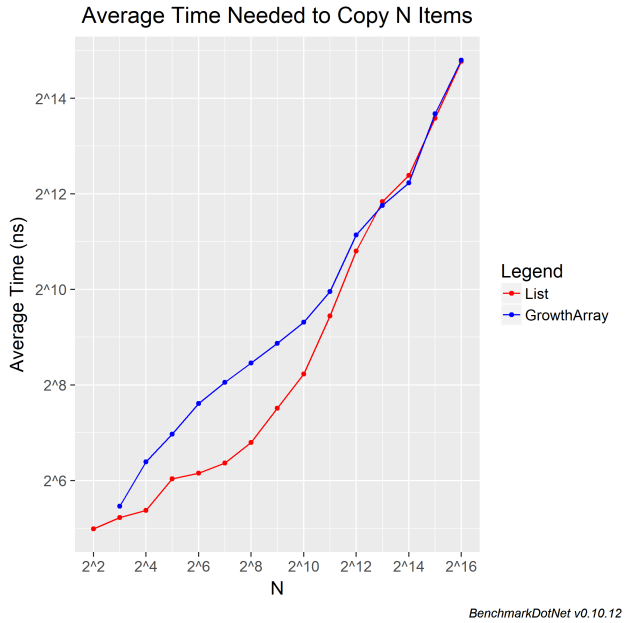

Figure 8: Time measurements for copying to raw arrays

Figures 4 and 5 show that growth arrays consistently outperform dynamic arrays at appending. Past $n = 4$, the blue line remains about one unit square below the red line for both graphs. Since the graphs are logarithmically scaled with one unit square representing a factor of 2, I conclude that growth arrays use approximately half the time and half the space to append the same number of items as dynamic arrays for values of n past 4.

It misleadingly looks like the lines are touching for values of n near powers of 2. At these points, the blue and red lines appear to make vertical "jumps." That is not what is really happening. The bottom end of each jump corresponds to when n is a power of 2, and the top end corresponds to when n is one more than a power of 2. The top end of each blue jump is close to the bottom end of each red jump, but it is incorrect to compare them since both points correspond to different values of n.

As you might guess, these jumps are where full capacity is reached and growth must occur. The sudden increase in time is mainly caused by the $O(n)$ memory allocation and (in dynamic arrays' case) the copying that happens when *Grow* is called. Notice that the blue jumps are much less drastic than the red ones; this is because growth arrays do not make a copy of n elements when they grow.

Figure 6 illustrates the graphs of 3 different random access algorithms, 2 of them for growth arrays. The constant time algorithms flat-line as they should, while the logarithmic time algorithm traces out a logarithmic curve. The dynamic array algorithm clearly beats both of the growth array algorithms: it appears to be approximately 8 times faster than the constant time algorithm I introduced, and around 4 times faster than the logarithmic time one at the point they are closest. An interesting observation is that the logarithmic time algorithm appears to outperform the blue constant time algorithm for values of n that are 128 or less.

Figures 7 and 8 show an interesting trend: the blue line converges towards the red line as n gets very

large. This supports the claim that I made in Sections 5.3 and 5.4 that not only do iteration and copying have the same time complexity for dynamic and growth arrays, their respective algorithms also have negligible time difference across data structures as n gets very large. Also worth noting is the large gap that opens between the red and blue lines in Figure 8 for values of n between 8 and 4096. This suggests that for small values of n, the extra overhead growth arrays introduce by iterating the tail's buffers is very significant.

7 Closing Remarks

In this paper, I formally developed the concept of growth arrays. By implementing several common dynamic array operations for them, I showed that they are a viable replacement for dynamic arrays. I proved that growth arrays outperform dynamic ones both time- and memory-wise when a large number of elements are appended. I exhibited logarithmic and constant time random access algorithms for growth arrays. Finally, I showed that they perform comparably to dynamic arrays for iteration and conversion to raw arrays.

Dynamic arrays are used frequently today because they are efficient at appending. Since growth arrays are even more efficient at it, they should be used in place of dynamic arrays when appending is performance-critical. Programming languages and frameworks should strongly consider adding support for growth arrays to improve program performance.

I will conclude this paper with several open-ended questions, which may serve as inspiration for further research.

- Can other array-based data structures, such as stacks or circular queues, also benefit from using growth arrays' algorithm for growth?
- Is there a constant time random access algorithm for growth arrays that does not use floating-point and works for all values of g and c_0?
- To what extent does growth arrays' poorer data locality result in performance losses? (Can tuning memory allocators offset these losses?)
- How can binary search and sorting algorithms be best implemented for growth arrays? (In particular, growth arrays with $g = 2$ have an evenly-divided structure: if 2^k items are appended, 2^{k-1} will be in the head and 2^{k-1} in the tail. Does this lend itself to a better binary search algorithm?)

Appendix

A Proofs of $\sim$ Properties

Note: In the following theorems and their proofs, variables with the letters f, g, or h denote functions that are positive for sufficiently large n.

A.1 $\sim$ is an Equivalence Relation

Theorem 4.1. *$\sim$ is reflexive, transitive, and symmetric.*

Proof. Clearly, $\lim_{n\to\infty} \frac{f}{f} = 1$, so $f \sim f$. Thus, $\sim$ is reflexive.

Suppose $f \sim g$. Then $\lim_{n\to\infty} \frac{f}{g} = 1$. Since both $\lim_{n\to\infty} 1$ and $\lim_{n\to\infty} \frac{f}{g}$ exist and the latter is nonzero, $\frac{\lim_{n\to\infty} 1}{\lim_{n\to\infty}(f/g)} = \lim_{n\to\infty} \frac{g}{f}$. Since the left-hand side evaluates to 1, $g \sim f$. Thus, $\sim$ is symmetric.

Suppose $f \sim g$ and $g \sim h$. By definition, $\lim_{n\to\infty} \frac{f}{g} = 1$ and $\lim_{n\to\infty} \frac{g}{h} = 1$. Since both limits exist, their product is $\lim_{n\to\infty} \left(\frac{f}{g} \cdot \frac{g}{h}\right) = \lim_{n\to\infty} \frac{f}{h}$. Since this product is 1, $f \sim h$. Thus, $\sim$ is transitive. □

A.2 $\sim$ Removes Lower-Order Terms

Theorem 4.2. *$f + o(f) \sim f$.*

Proof. Let $g = o(f)$. By definition, $\lim_{n\to\infty} \frac{g}{f} = 0$. Since $\lim_{n\to\infty} \frac{f}{f}$ and $\lim_{n\to\infty} \frac{g}{f}$ both exist,

$$\lim_{n\to\infty} \frac{f+g}{f} = \lim_{n\to\infty} \frac{f}{f} + \lim_{n\to\infty} \frac{g}{f} = 1 + 0 = 1$$

It follows that $f + g = f + o(f) \sim f$. □

A.2.1 $f + c \sim f$ for Unbounded f

Corollary 4.2.1. *If f is unbounded, then $f + c \sim f$ for any constant c.*

Proof. If f is unbounded, then $c = o(f)$. Applying Theorem 4.2, $f + c \sim f$. □

A.3 $\sim$ Respects $+$, $\times$, and $\div$

Theorem 4.3. *If $f \sim f_0$ and $g \sim g_0$, then*

$$f + g \sim f_0 + g_0$$
$$fg \sim f_0 g_0$$
$$\frac{f}{g} \sim \frac{f_0}{g_0}$$

Proof. Let $q = \frac{f+g}{f_0+g_0}$, and let $d = g_0 + g_0^2/f_0$. q may be expressed as

$$\begin{aligned} q &= \frac{(g_0/f_0)(f+g)}{(g_0/f_0)(f_0+g_0)} \\ &= \frac{g_0(f/f_0) + g_0 g/f_0}{d} \\ &= \frac{g_0(f/f_0) + (g_0^2/f_0)(g/g_0)}{d} \\ &= \frac{g_0}{d} \cdot \frac{f}{f_0} + \frac{g_0^2/f_0}{d} \cdot \frac{g}{g_0} \end{aligned}$$

Now, consider that

$$\begin{aligned} q - 1 = q - \frac{d}{d} &= q - \frac{g_0}{d} - \frac{g_0^2/f_0}{d} \\ &= \frac{g_0}{d} \cdot \frac{f}{f_0} + \frac{g_0^2/f_0}{d} \cdot \frac{g}{g_0} - \frac{g_0}{d} - \frac{g_0^2/f_0}{d} \\ &= \frac{g_0}{d} \cdot \left(\frac{f}{f_0} - 1\right) + \frac{g_0^2/f_0}{d} \cdot \left(\frac{g}{g_0} - 1\right) \\ \lim_{n\to\infty}(q-1) &= \lim_{n\to\infty} \left(\frac{g_0}{d} \cdot \left(\frac{f}{f_0} - 1\right) + \frac{g_0^2/f_0}{d} \cdot \left(\frac{g}{g_0} - 1\right)\right) \\ &= \lim_{n\to\infty} \left(\frac{g_0}{d} \cdot \left(\frac{f}{f_0} - 1\right)\right) + \lim_{n\to\infty} \left(\frac{g_0^2/f_0}{d} \cdot \left(\frac{g}{g_0} - 1\right)\right) \end{aligned}$$

Since g_0 and g_0^2/f_0 sum to d and all functions are positive, $\frac{g_0}{d}$ and $\frac{g_0^2/f_0}{d}$ are bounded between 0 and 1 as $n \to \infty$. From the given, $\lim_{n\to\infty}\left(\frac{f}{f_0} - 1\right)$ and $\lim_{n\to\infty}\left(\frac{g}{g_0} - 1\right)$ both exist and equal 0. The limit of a bounded expression times one approaching 0 is 0; thus, both limits on the right-hand side are 0.

Substituting, I receive $\lim_{n\to\infty}(q-1) = 0$, so $\lim_{n\to\infty} q = 1$. Since q is defined as $\frac{f+g}{f_0+g_0}$, this shows that $f + g \sim f_0 + g_0$, proving the theorem statement for addition.

The statement is proven much more easily for multiplication. Since both $\lim_{n\to\infty} \frac{f}{f_0}$ and $\lim_{n\to\infty} \frac{g}{g_0}$ exist,

$$\lim_{n\to\infty} \frac{fg}{f_0 g_0} = \left(\lim_{n\to\infty} \frac{f}{f_0}\right)\left(\lim_{n\to\infty} \frac{g}{g_0}\right) = 1 \cdot 1 = 1$$

It follows that $fg \sim f_0 g_0$. If the limit for g is flipped before multiplying, the following results:

$$\lim_{n\to\infty} \frac{f/g}{f_0/g_0} = 1$$

This implies that $\frac{f}{g} \sim \frac{f_0}{g_0}$. □

A.3.1 $\sim$ Equations can be Algebraically Manipulated

Corollary 4.3.1. *A $\sim$ relation is preserved when both sides are added, multiplied, or divided by the same positive quantity.*

Proof. This immediately follows from Theorem 4.3 by taking $g_0 = g$. □

A.4 Asymptotic Functions Have the Same Big-O Class

Theorem 4.4. *If $f \sim g$ and $g = O(h)$, then $f = O(h)$.*

Proof. Let $r = \frac{f}{g}$. Since $\lim_{n\to\infty} r = 1$, $r \leq 2$ for sufficiently large n. By definition, $\exists c : g \leq ch$ for sufficiently large n, where c is a positive constant. Then, $f = rg \leq rch \leq 2ch$ for sufficiently large n. Thus, $f = O(h)$. □

A.5 Asymptotic Functions may be Interchanged in $\prec$ and $\preceq$ Inequalities

Theorem 4.5. *If $f \sim f_0$ and $f_0 \prec f_1$, then $f \prec f_1$. If $g \sim g_1$ and $g_0 \prec g_1$, then $g_0 \prec g$.*
These statements also hold for $\preceq$.

Proof. By definition, $\lim_{n\to\infty} \frac{f}{f_0} = 1$ and $\lim_{n\to\infty} \frac{f_0}{f_1} < 1$. Since both limits exist,

$$\lim_{n\to\infty} \frac{f}{f_1} = \left(\lim_{n\to\infty} \frac{f}{f_0}\right)\left(\lim_{n\to\infty} \frac{f_0}{f_1}\right) < 1 \cdot 1 = 1$$

It follows that $f \prec f_1$.

The second statement may be proved in a similar manner. Since $\sim$ is symmetric, $g_1 \sim g$. Applying the same argument as before,

$$\lim_{n\to\infty} \frac{g_0}{g} = \left(\lim_{n\to\infty} \frac{g_0}{g_1}\right)\left(\lim_{n\to\infty} \frac{g_1}{g}\right) < 1 \cdot 1 = 1$$

Thus, $g_0 \prec g$.

From these results and the transitivity of $\sim$, it can trivially be shown that the same statements are true when $\prec$ is replaced with $\preceq$. □

A.6 $\prec$ and $\preceq$ Inequalities can be Algebraically Manipulated

Theorem 4.6. *A $\prec$ inequality is preserved when both sides are multiplied or divided by the same positive quantity.*
This statement also holds for $\preceq$.

Proof. Suppose that $f \prec g$. By definition, $\lim_{n\to\infty} \frac{f}{g} < 1$. For any positive function h, $\lim_{n\to\infty} \frac{fh}{gh} < 1$ and $\lim_{n\to\infty} \frac{f/h}{g/h} < 1$, so respectively $fh \prec gh$ and $\frac{f}{h} \prec \frac{g}{h}$.

From these results and Corollary 4.3.1, it can trivially be shown that the same statement is true when $\prec$ is replaced with $\preceq$. □

A.6.1 $\prec$ and $\preceq$ Inequalities are Flipped when Both Sides are Inverted

Corollary 4.6.1. *If $f \prec g$, then $\frac{1}{g} \prec \frac{1}{f}$.*
This statement also holds for $\preceq$.

Proof. This immediately follows from $f \prec g$ or $f \preceq g$ by using Theorem 4.6 to divide both sides by fg. □

A.7 Same-Order Terms may be Added to Both Sides of $\prec$ and $\preceq$ Inequalities

Theorem 4.7. *Suppose $f \prec g$. If $\lim_{n\to\infty} \frac{f}{h} > 0$ and $g = O(h)$, then $f + h \prec g + h$.*
This statement also holds for $\preceq$.

Proof. Assume that n is arbitrarily large. Since $\frac{f}{h}$ has a positive limit, it is bounded below by some positive constant c_1. Rearranging, $c_1 h \leq f$. From $f \prec g$, it follows that $f < g$ for sufficiently large n. Since $g = O(h)$, $\exists c_2 : g \leq c_2 h$. Combining these inequalities, I receive $c_1 h \leq f < g \leq c_2 h$. From this, it is clear that $c_1 < c_2$.

Now, consider that

$$\lim_{n\to\infty} \frac{f+h}{g+h} \leq \lim_{n\to\infty} \frac{c_1 h + h}{c_2 h + h} = \frac{c_1 + 1}{c_2 + 1}$$

Since $c_1 < c_2$, $c_1 + 1 < c_2 + 1$ so $\frac{c_1+1}{c_2+1} < 1$. Thus, $f + h \prec g + h$.

From these results and Corollary 4.3.1, it can trivially be shown that the same statement is true when $\prec$ is replaced with $\preceq$. □

References

[1] de Bruijn, Nicolaas G. *Asymptotic Methods in Analysis*. Dover Publications, 1981.

[2] Cormen et al. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.

[3] Fiebrink, Rebecca. *Amortized Analysis Explained*. Princeton University, 2007. Available at `https://www.cs.princeton.edu/~fiebrink/423/AmortizedAnalysisExplained_Fiebrink.pdf`.

[4] Fog, Agner. *Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*. Technical University of Denmark, 2017. Available at `http://www.agner.org/optimize/instruction_tables.pdf`.

[5] Leiserson et al. *Using de Bruijn sequences to Index a 1 in a Computer Word*. MIT Laboratory for Computer Science, 1998. Available at `http://supertech.csail.mit.edu/papers/debruijn.pdf`.