# Lecture Notes on Amortized Analysis

15-122: Principles of Imperative Computation
Rob Simmons, Frank Pfenning

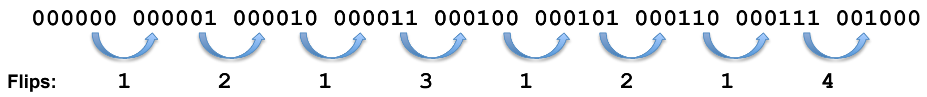
Lecture 12
October 7, 2014

## 1 Introduction

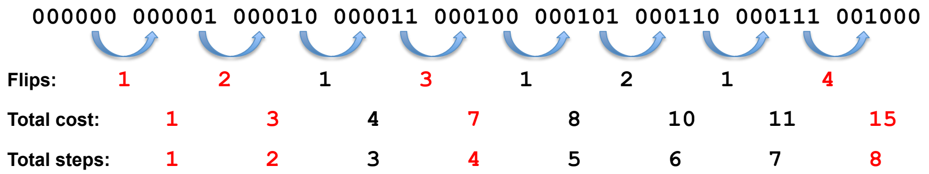
Most lectures so far had topics related to all three major categories of learning goals for the course: computational thinking, algorithms, and programming. The same is true for this lecture. With respect to algorithms, we introduce *unbounded arrays* and operations on them. Analyzing them requires *amortized analysis*, a particular way to reason about sequences of operations on data structures. We also briefly talk about again about *data structure invariants* and *interfaces*, which are crucial computational thinking concepts.

## 2 The $k$-bit Counter

A simple example we use to illustrate amortized analysis is the idea of a *binary counter* that we increment by one at a time. If we have to flip each bit individually, flipping $k$ bits takes $O(k)$ time.

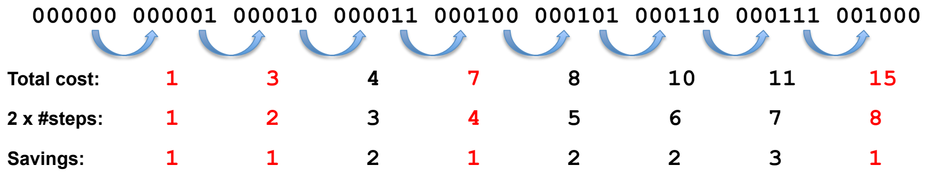


Obviously, if we have a $k$-bit counter, the worst case running time of an single increment operation is $O(k)$. But does it follow that the worst case running time of $n$ operations is $O(kn)$? Not necessarily. Let's look more carefully at the cases where the operation we have to perform is the *most* *expensive operation we've yet considered:*



We can observe two things informally. First, the most expensive operations get further and further apart as time goes on. Second, whenever we reach a most-expensive-so-far operation at step $n$, the total cost of all the operations up to and including that operation is $2n - 1$. Can we extend this reasoning to say that the total cost of performing $n$ operations will never exceed $2n$?

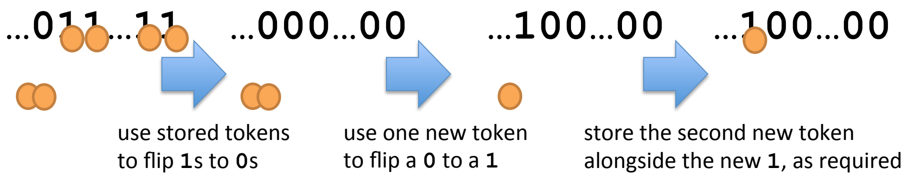
One metaphor we frequently use when doing this kind of analysis is banking. It's difficult to think in terms of savings accounts full of microseconds, so when we use this metaphor we usually talk about *tokens*, representing an abstract notion of cost. With a token, we can pay for the cost of a particular operation; in this case, the constant-time operation of flipping a bit. If we *reserve (or budget) two tokens* every time we perform any increment, putting any excess into a savings account, then we see that after the expensive operations we've looked out, our savings account contains 1 token. Our savings account appears to never run out of money.



This is good evidence, but it still isn't a proof. To offer something like proof, as always, we need to talk in terms of *invariants*. And we can see a very useful invariant: the number of `1` bits always matches the number in our savings account! This observation leads us to the last trick that we'll use when we perform amortized analysis in this class: we associate one token with each `1` in the counter as *part of our data structure invariant*.

## 3 Amortized Analysis With Data Structure Invariants

Whenever we increment the counter, we'll always flip some number (maybe zero) lower-order `1`s to `0`, and then we'll flip a single `0` to `1` (unless we're out of bits in the counter). No matter how many lower-order `1` bits there are, the flipping of those low-order bits is paid for by the token associated with those bits. Then, because we're always gaining 2 more tokens whenever we perform an increment, one of those tokens can be used to flip the lowest-order `0` to a `1` and the other one can be associated with that new `1` in order to make sure the data structure invariant is preserved. Graphically, *any* time we increment the counter, it looks like this:



(Well, not every time: if the counter is limited to $k$ bits and they're all `1`, then we'll flip all the bits to `0`. In this case, we can just throw away or lose track of our two new tokens, because we can restore the data structure invariant without needing the two new tokens. In the accounting or banking view, when this happens we observe that our savings account now has some extra savings that we'll never need.)

Now that we've rephrased our operational argument about the size of savings as data structure invariant that is always preserved by the increment operation, we can securely say that, each time we increment the counter, we need to reserve exactly two tokens. This means that a series of $n$ increments of the $k$-bit counter, starting will the counter is all zeroes, will take time in $O(n)$. We can also say that each individual operation has an *amortized running time* of 2 bitflips, which means that the amortized cost is in $O(1)$. It's not at all contradictory for bitflips to have an amortized running time in $O(1)$ and a worst-case running time in $O(k)$.

In summary: to talk about amortized running time (or, more generally, the amortized *cost*) of operations on a data structure, we:

1. Invent a notion of *tokens* that stand in for the resource that we're interested in (usually time);

2. Specify, for any instance of the data structure, how many tokens need to be held in reserve as part of the data structure invariant;

3. Assign, for each for operation we might perform on the data structure, an amortized cost in tokens;

4. Prove that, for any operation we might perform on the data structure, the amortized cost plus the tokens held in reserve as part of the data structure invariant suffices to restore the data structure invariant.

This analysis proves that, for any sequence of operations on a data structure, the cumulative cost of that sequence of operations will be less than the sum of the amortized cost of those operations. Even if some of the operations in that sequence have high cost (take a long time to run), that will be at least paid for by other operations that have low cost (take a short time to run).

This form of amortized analysis is sometimes called the *potential method*. It is a powerful mathematical technique, but we'll only use it for relatively simple examples in this class.

## 4 Unbounded Arrays

In the second homework assignment, you were asked to read in some files such as the *Collected Works of Shakespeare*, the *Scrabble Players Dictionary*, or anonymous tweets collected from Twitter. What kind of data structure do we want to use when we read the file? In later parts of the assignment we want to look up words, perhaps sort them, so it is natural to want to use an array of strings, each string constituting a word. A problem is that before we start reading we don't know how many words there will be in the file so we cannot allocate an array of the right size! One solution uses either a queue or a stack.

The array interface that we originally introduced in Lecture 9 wouldn't work, because it requires us to bound the size of the array – to know in advance how much data we'll need to store:

```
typedef struct arr_header* arr; // typedef _______ arr;
int    arr_len(arr A);
arr    arr_new(int size)
  /*@requires 0 <= size; @*/
  /*@ensures arr_len(\result) == size; @*/;
string arr_get(arr A, int i)
  /*@requires 0 <= i && i < arr_len(A); @*/;
void   arr_set(arr A, int i, string x)
  /*@requires 0 <= i && i < arr_len(A); @*/;
```

It would work, however, if we had an extended interface of *unbounded arrays*, where the `arr_add(A,x)` function appends `x` to the array index that was previous past the end of the array and incrementing the array's size so that this index is now in-bounds. There's a complementary operation, `arr_rem(A)`, that decreases the array's size by 1.

```
void   arr_add(arr A, string x);
void   arr_rem(arr A) /*@requires 0 < arr_len(A); @*/;
```

We'd like to give all the operations in this extended array interface a running time in $O(1)$.[1] It's not practical to give `arr_add(A,x)` a worst case running time in $O(1)$, but with a careful implementation we can show that is possible to give the function an amortized running time in $O(1)$.

## 5 Implementing Unbounded Arrays

Our original implementation of an interface for C0 arrays had a struct with two fields: the `data` field, an actual array of strings, and a `limit` field, which contained the length of the array. This limit was what we returned to the user when they asked for the length of the array.

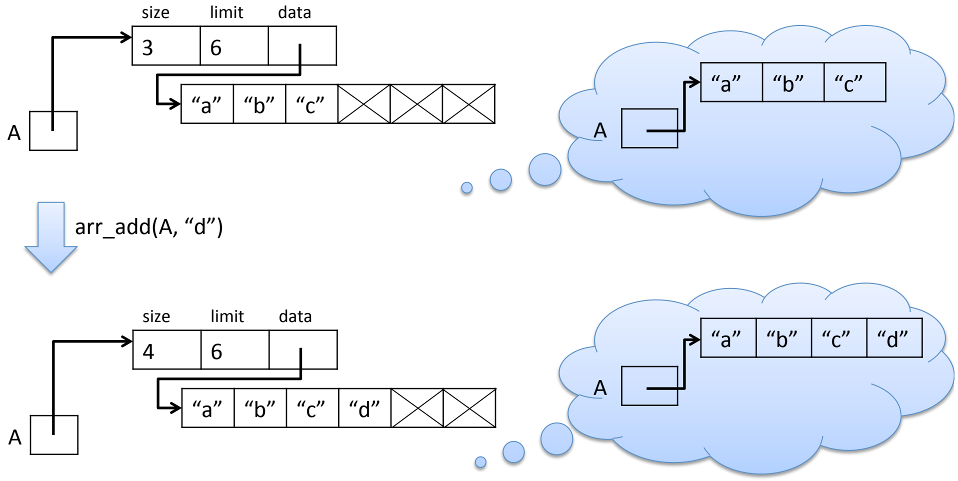
While it wouldn't work to have a limit that was *less* than the array length we are reporting to the user, we can certainly have an array limit that is greater than the length we're report to the user: we'll store this smaller number in the `size` field.

```
typedef struct arr_header* arr;
struct arr_header {
  int size;                /* 0 <= size && size < limit */
  int limit;               /* 0 < limit */
  string[] data;           /* \length(data) == limit */
};

int arr_len(arr A)
//@requires is_arr(A);
//@ensures 0 <= \result && \result <= \length(A->data);
{
  return A->size;
}
```

If we reserve enough extra room, then most of the time when we need to use `arr_add` to append a new item onto the end of the array, we can do

[1] It's questionable at best whether we should think about `arr_new` being $O(1)$, because we have to allocate $O(n)$ space to get an array of length $n$ and initialize all that space to default values. The operating system has enough tricks to get this cost down, however, that we usually think of array allocation as a constant-time operation.

it by just incrementing the size field and putting the new element into an already-allocated cell in the data array.



The images to the left above represent how the data structure is actually stored in memory, and the images in the thought bubbles to the right represent how the client of our array library can think about the data structure and after a arr_add operation.

The data structure invariant sketched out in comments above can be turned into an is_arr function like this:

```
bool is_arr_expected_length(string[] A, int limit) {
  //@assert \length(A) == limit;
  return true;
}

bool is_arr(struct arr_header* AH) {
  return AH != NULL
    && is_arr_expected_length(AH->data, AH->limit)
    && 0 <= AH->size && AH->size < AH->limit;
}
```

Because we require that the size is strictly less than the limit, we can always implement arr_add by storing the new string in A->data[A->size] and then incrementing the size. But after incrementing the size, we might violate the data structure invariant! We'll use a helper function, arr_resize, to resize the array in this case.

```
void arr_add(arr A, string x)
//@requires is_arr(A);
//@ensures is_arr(A);
{
  A->data[A->size] = x;
  (A->size)++;
  arr_resize(A);
}
```

The arr_resize() function works by allocating a new array, copying the old array's contents into the new array, and replacing A->data with the address of the newly allocated array.

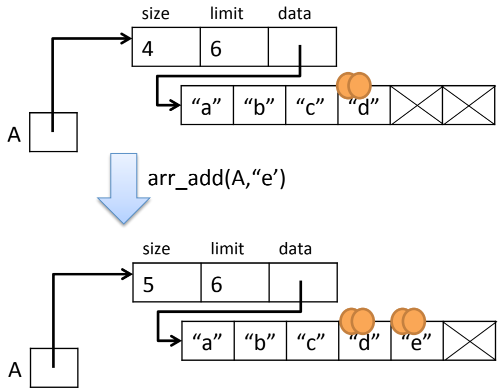
```
void arr_resize(arr A)
//@requires A != NULL && \length(A->data) == A->limit;
//@requires 0 < A->size && A->size <= A->limit;
//@ensures is_arr(A);
{
  if (A->size == A->limit) {
    assert(A->limit < int_max() / 2); // Can't handle bigger
    A->limit = A->size * 2;

  } else {
    return;
  }

  //@assert 0 <= A->size && A->size < A->limit;
  string[] B = alloc_array(string, A->limit);

  for (int i = 0; i < A->size; i++)
  //@loop_invariant 0 <= i && i <= A->size;
  {
    B[i] = A->data[i];
  }

  A->data = B;
}
```

The assertion assert(A->limit < int_max() / 2) is there because, without it, we have to worry that doubling the limit in the next line might overflow. *Hard asserts* like this allow us to safely detect unlikely failures that we can't exclude with contracts.

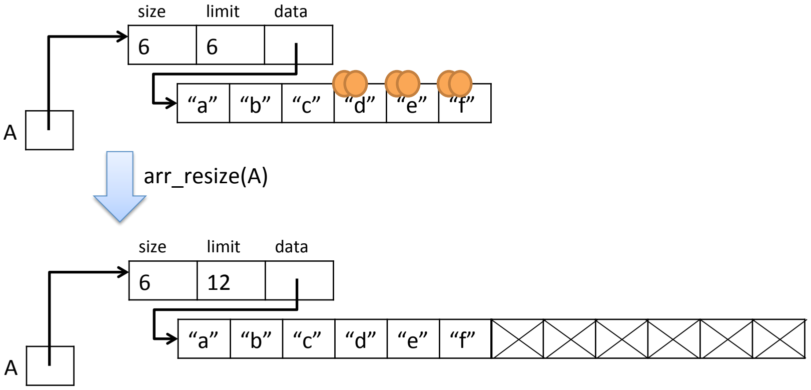
## 6 Amortized Analysis for Unbounded Arrays

Doubling the size of the array whenever we resize it allows us to give an amortized analysis concluding that every `arr_add` operation has an amortized cost of *three* array writes. Because array writes are our primary notion of cost, we say that one token allows us to write to an array one time.

Our data structure invariant for tokens is that, whenever we are using a cell in the *second half of the array*, we need to store two tokens alongside that cell. Every call to `arr_add` uses one token to write an element into the array; if that new element is in the second half of the array, we store two tokens. alongside that newly-in-use cell. Thus, budgeting three tokens for each `arr_add` operation suffices to preserve the data structure invariant in every case that doesn't cause the array to become totally full.



In the cases where the addition does completely fill the array, we need to copy over every element in the old array into a new, larger array in order to preserve the `A->size < A->limit` data structure invariant. This requires one write for every element in the old array. We can pay for each one of those writes because we have two stored tokens in exactly half of the old array – which is the same as having one token for each cell in the old array.



After the resize, exactly half the array is full, so our data structure invariant for tokens doesn't require us to have any tokens in reserve. This means that the data structure invariant is preserved in this case as well.

This establishes that the amortized cost of `arr_add` is three array writes. We do things that aren't array writes in the process of doing `arr_add`, but the cost is dominated by array writes, so this gives the right big-O notion of (amortized) cost.

## 7 Shrinking the array

In the example above, we only re-sized our array to make it bigger. We could also call `arr_resize(A)` in our `arr_rem` function, and allow that function to make the array either bigger or smaller.

```
void arr_rem(arr A)
//@requires is_arr(A);
//@requires 0 < arr_len(A);
//@ensures is_arr(A);
{
  (A->size)--;
  arr_resize(A);
}
```

If we want `arr_rem` to take amortized constant time, it will not work to resize the array when `A` is less than half full. An array that is exactly half full doesn't have any tokens in reserve, so it wouldn't be possible to pay for halving the size of the array in this case. In order to make the constant-time amortized cost work, the easiest thing to do is only resize the array when it is less than *one-quarter* full. If we make this change, it's possible to reflect it in the data structure invariant, requiring that `A->size` be in the range [`A->limit/4`, `A->limit`) rather than the range [0, `A->limit`) that we required before.

In order to show that this deletion operation has the correct amortized cost, we must extend our data structure invariant to . (See the exercises below.) Once we do so, we can conclude that *any* valid sequence of $n$ operations (`arr_add` or `arr_rem`) that we perform on an unbounded array will take time in $O(n)$, even if any single one of those operations might take time proportional to the current length of the array.

## Exercises

**Exercise 1** *If we only add to an unbounded array, then we'll never have less than half of the array full. If we want* `arr_rem` *to be able to make the array smaller, we'll need to reserve tokens when the array is less than half full, not just when the array is more than half full. What is the precise data structure invariant we need? How many tokens (at minimum) do we need to per* `arr_rem` *operation in order to preserve it? What is the resulting amortized cost (in terms of array writes) of* `arr_rem`*?*

**Exercise 2** *If we also said that we required $n$ tokens to allocate an array of size $n$, then the* `arr_new` *function would obviously have an cost (amortized and worst-case) of $2n \in O(n)$. How many tokens would we need to budget for each* `arr_add` *and* `arr_rem` *operation in order to prove that these operations require an amortized constant number of tokens?*

**Exercise 3** *How would our amortized analysis change if we increased the size of the array by 50% instead of 100%? What if we increased it by 300%? You are allowed to have a cost in fractions of a token.*

**Exercise 4** *When removing elements from the unbounded array we resize if the limit grossly exceeds its size. Namely when* `L->size < L->limit/4`*. Your first instinct might have been to already shrink the array when* `L->size < L->limit/2`*. We have argued by example why that does not give us constant amortized cost $O(n)$ for a sequence of $n$ operations. We have also sketched an argument why* `L->size <= L->limit/2` *gives the right amortized cost. At which step in that argument would you notice that* `L->size <= L->limit/2` *is the wrong choice?*