Lecture 11
Unbounded Arrays

15-122: Principles of Imperative Computation (Spring 2024)
Rob Simmons, Frank Pfenning

Arrays have efficient $O(1)$ access to elements given an index, but their size is set at allocation time. This makes storing an unknown number of elements problematic: if the size is too small we may run out of places to put them, and if it is too large we will waste memory. Linked lists do not have this problem at all since they are extensible, but accessing an element is $O(n)$. In this lecture, we introduce *unbounded arrays*, which like lists can hold an arbitrary number of elements, but also allow these element to be retrieved in $O(1)$ time? What gives? Adding (and removing) an element to the unbounded array has cost either $O(1)$ or $O(n)$, but in the long run the average cost of each such operation is constant — the challenge will be to prove this last statement!

Additional Resources

- Review slides (https://cs.cmu.edu/~15122/handouts/slides/review/11-uba.pdf)
- Code for this lecture (https://cs.cmu.edu/~15122/handouts/code/11-uba.tgz)
- There is one short video associated with this lecture:
  - Amortized Analysis (https://youtu.be/L8cXZ_4RHt8)

This maps to our learning goals as follows

Programming: We introduce *unbounded arrays* and operations on them.

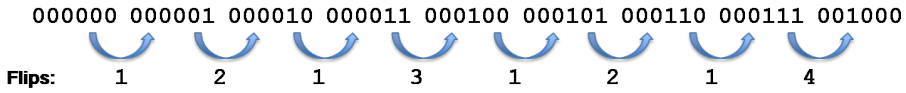
Algorithms and Data Structures: Analyzing them requires *amortized analysis*, a particular way to reason about sequences of operations on data structures.

Computational Thinking: We also briefly talk again about *data structure invariants* and *interfaces*, which are crucial computational thinking concepts.

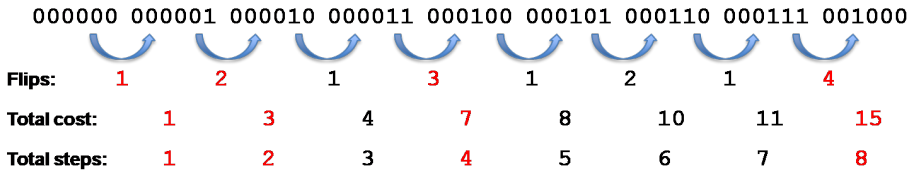
But first, let's introduce the idea of amortized analysis on a simpler example.

1 The n-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 n bits takes $O(n)$ time.

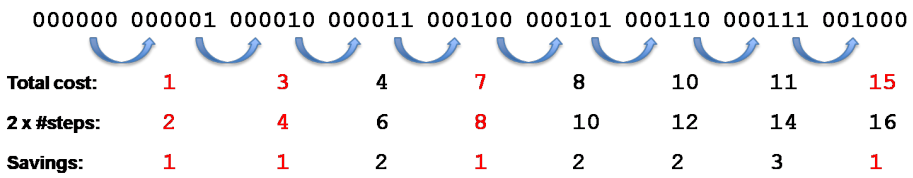


Obviously, if we have an n-bit counter, the worst case running time of an single increment operation is $O(n)$. But does it follow that the worst case running time of k operations is $O(kn)$? Not necessarily. Let's look more carefully at the cases where the operation we have to perform is the *most expensive 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 k, the total cost of all the operations up to and including that operation is $2k - 1$. Can we extend this reasoning to say that the total cost of performing k operations will never exceed $2k$?

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.

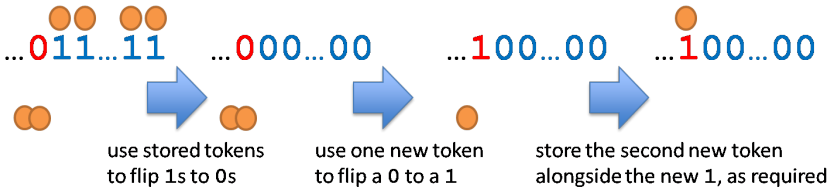


This is good evidence, but it still isn't a proof. To offer something like a proof, as always, we need to talk in terms of *invariants*. And we can see a very useful invariant: the number of `1` bits always matches the number in our savings account! This observation leads us to the last trick that we'll use when we perform amortized analysis in this class: we associate one token with each `1` in the counter as part of a *meta data structure invariant*. Like normal data structure invariants, this meta data data structure invariant should hold before and after carrying out an operation on the data structure. Differently from normal data structure invariants, it is not captured in code — it resides in our minds only.

2 Amortized Analysis With Data Structure Invariants

Whenever we increment the counter, we'll always flip some number (maybe zero) of lower-order `1`'s to `0`, and then we'll flip exactly one `0` to `1` (unless we're out of bits in the counter). For example, incrementing the 8-bit counter `10010011` yields `10010100`: the two rightmost `1`'s got flipped to `0`'s, the rightmost `0` was flipped into a `1`, and all other bits were left unchanged. We can explain budgeting two tokens for each increment as follows: one token is used to pay for flipping the rightmost `0` in the current operation, and one token is saved for when the resulting `1` will need to be flipped into a `0` as part of a future increment. So, how do we pay for flipping the rightmost `1`'s in this example? By using the tokens that were saved when they got flipped from a `0` into the current `1`.

No matter how many lower-order `1` bits there are, the flipping of those low-order bits is paid for by the tokens associated with those bits. Then, because we're always gaining 2 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 n 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 amount of savings as a data structure invariant that is always preserved by the increment operation, we can securely say that, each time we increment the counter, it suffices to reserve exactly two tokens. This means that a series of k increments of the n-bit counter, starting when the counter is all zeroes, will take time in $O(k)$. We can also say that each individual operation has an *amortized running time* of 2 bit flips, which means that the *amortized* cost of each operation is in $O(1)$. It's not at all contradictory for bit flips to have an amortized running time in $O(1)$ and a worst-case running time in $O(n)$.

In summary: to talk about the 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 — in our example, a token is spent each time a bit is flipped);
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 (in our example, one token for each 1-bit);
3. Assign, for each operation we might perform on the data structure, an amortized cost in tokens (in our example, two tokens for each increment);
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 or equal to 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).

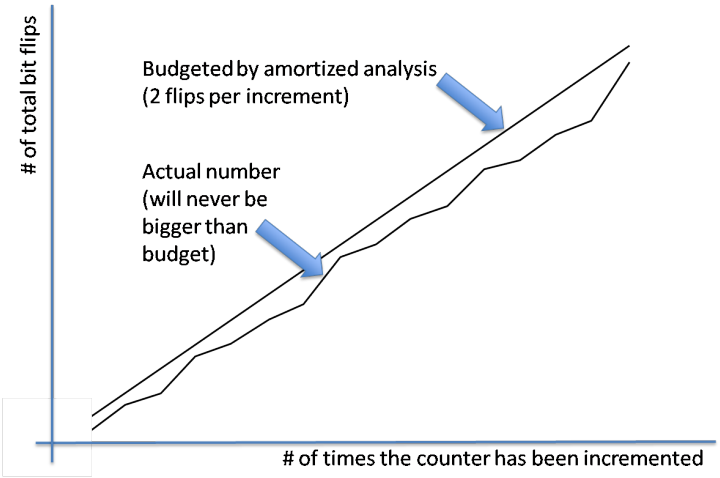
This form of amortized analysis is sometimes called the *potential method*. It is a powerful mathematical technique, but we'll only use it for relatively simple examples in this class.

3 What amortized analysis means

Tokens aren't real things, of course! They are stand-ins for the actual resources we're interested in. Usually, the resource we are concerned about

is time, so we match up tokens to the (frequently constant-time) operations we have to do on our data structure. In the current example, we might be storing the counter as an array of **`bool`** values, in which case it would take a constant-time array write to flip one of the bits in the counter. (Tokens will also correspond to array writes in the unbounded array example we consider next.)

We do amortized analysis in order to prove that the *inexpensive operations early on* suffice to pay for *any expensive operations* that happen later. There's no uncertainty with amortized analysis: we know that, if we calculate our overall time as if each increment costs two bit flips, we will never underestimate the total cost of our computation.



This is different than average case analysis for quicksort, where we know that sometimes the total cost of sorting could be higher than predicted (if we get unlucky in our random pivot selection). There's no luck in our amortized analysis: we know that the total cost of k increments is in $O(k)$, even though the worst case cost of a single increment operation is $O(n)$ bit flips.

4 Unbounded Arrays

In a previous 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.

A non-sorting variant of the self-sorting array interface that we discussed before doesn't seem like it would work, because it requires us to bound the size of the array — to know in advance how much data we'll need to store. Let's call this unsorted variant `uba_t` and rename the rest of the interface accordingly:

```
// typedef ______* uba_t;

int uba_len(uba_t A)
  /*@requires A != NULL;   @*/
  /*@ensures \result >= 0; @*/ ;

uba_t uba_new(int size)
  /*@requires 0 <= size;                @*/
  /*@ensures \result != NULL;           @*/
  /*@ensures uba_len(\result) == size;  @*/;

string uba_get(uba_t A, int i)
  /*@requires A != NULL;                @*/
  /*@requires 0 <= i && i < uba_len(A); @*/;

void uba_set(uba_t A, int i, string x)
  /*@requires A != NULL;                @*/
  /*@requires 0 <= i && i < uba_len(A); @*/;
```

It would work, however, if we had an extended interface of *unbounded arrays*, where the `uba_add(A,x)` function increases the array's size to add `x` to the end of the array. There's a complementary operation, `uba_rem(A)`, that decreases the array's size by 1.

```
void uba_add(uba* A, string x)
  /*@requires A != NULL;      @*/;

string uba_rem(uba* A)
  /*@requires A != NULL;      @*/
  /*@requires 0 < uba_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 `uba_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

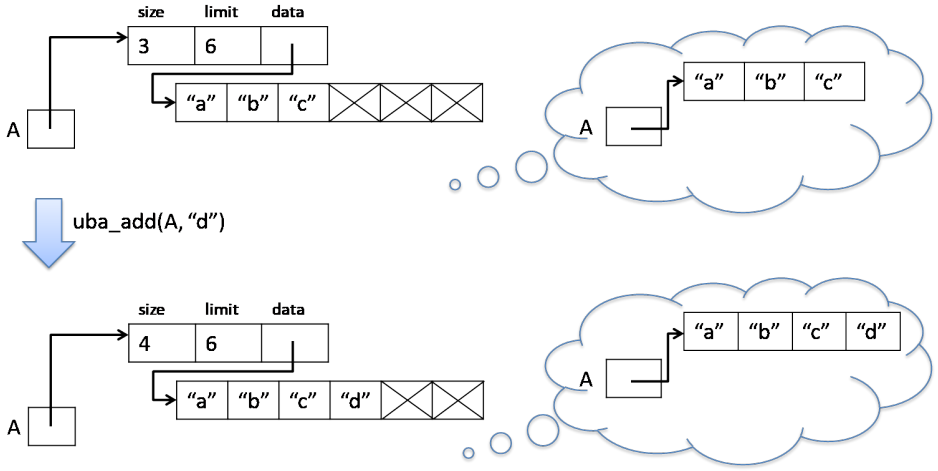
Our original implementation of an interface for self-sorting arrays had a struct with two fields: the `data` field, an actual array of strings, and a `length` field, which contained the length of the array. This value, which we will call the *limit* when talking about unbounded arrays, was what we returned to the users 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: we'll store the potentially smaller number that we report in the `size` field.

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

int uba_len(uba* A)
//@requires is_uba(A);
//@ensures 0 <= \result && \result <= \length(A->data);
{
  return A->size;
}
```

[1] It's questionable at best whether we should think about `uba_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.

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

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

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

bool is_uba(uba* A) {
  return A != NULL
    && 0 <= A->size && A->size < A->limit
    && is_uba_expected_length(A->data, A->limit);
}
```

Because we require that the size be strictly less than the limit, we can always implement `uba_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, `uba_resize`, to resize the array in this case.

```
void uba_add(uba* A, string x)
```

```
//@requires is_uba(A);
//@ensures is_uba(A);
{
  A->data[A->size] = x;
  (A->size)++;
  uba_resize(A);
}
```

The `uba_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.

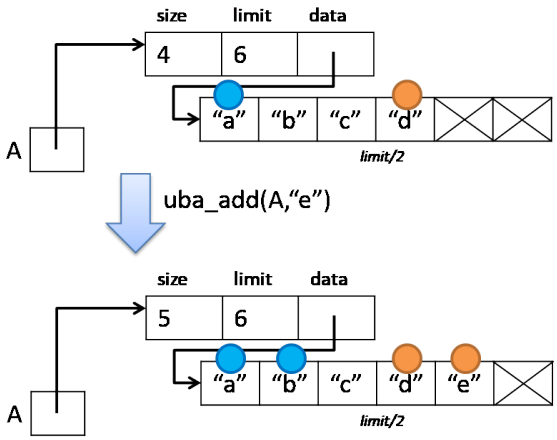
```
void uba_resize(uba* A)
//@requires A != NULL && \length(A->data) == A->limit;
//@requires 0 < A->size && A->size <= A->limit;
//@ensures is_uba(A);
{
  if (A->size == A->limit) {
    assert(A->limit <= int_max() / 2); // Can't handle bigger
    A->limit = A->size * 2;
  } else {
    return;
  }

  //@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 but that we don't want to encode into our interface.

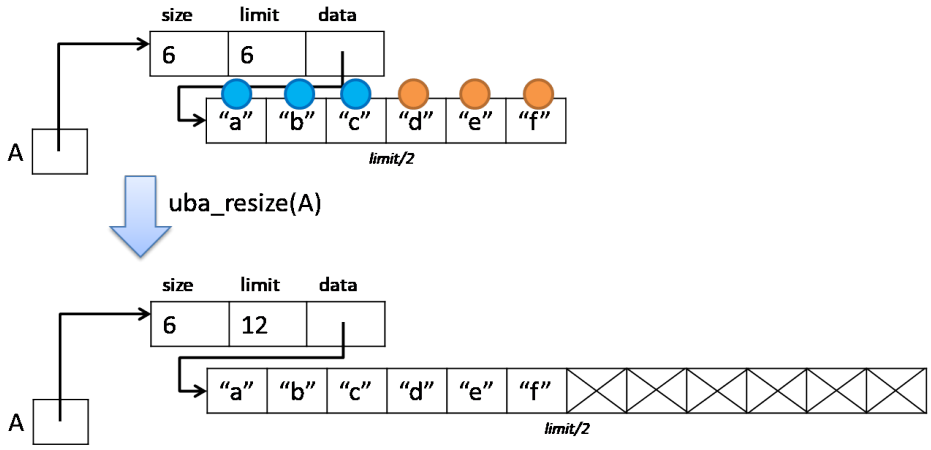
6 Amortized Analysis for Unbounded Arrays

Doubling the size of the array whenever we resize it allows us to give an amortized analysis concluding that every `uba_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.

Here's how the analysis works: our *data structure invariant* for tokens is that every cell in use in the *second half of the array* (i.e., each cell at index i in $[limit/2, size)$) and the corresponding cell in the first half of the array (at index $i - limit/2$) will have one token associated with it. For clarity, we color tokens in the first half of the array in blue and tokens in the second half in gold. Each time we add an element to the array (at index $size$), we use one token to perform that very write, store one gold token with this element for copying it next time we double the size the array in a future resize, and store one blue token for copying the corresponding element in the first half of the array (at index $size - limit/2$) on that same resize. Thus, budgeting three tokens for each `uba_add` operation suffices to preserve the data structure invariant in every case that doesn't cause the array to become totally full. We therefore assign an *amortized cost* of three tokens to the add operation.



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 now have one gold token for each element in the second half of the old array (at indices $[limit/2, limit)$) and one blue token for each element in the first half of that array (at indices $[0, limit/2)$) — 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. In general, the number of tokens associated with an unbounded array is $2 \times size - limit$.

This establishes that the amortized cost of `uba_add` is three array writes. We do things that aren't array writes in the process of doing `uba_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 resized our array to make it bigger. We could also call `uba_resize(A)` in our `uba_rem` function, and allow that function to make the array either bigger or smaller.

```
string uba_rem(uba* A)
//@requires is_uba(A);
//@requires 0 < uba_len(A);
//@ensures is_uba(A);
{
  (A->size)--;
  string x = A->data[A->size];
  uba_resize(A);
  return x;
}
```

If we want `uba_rem` to take amortized constant time, it will not work to resize the array as soon as `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 also store tokens for every unused cell in the left half of the array. (See the exercises below.) Once we do so, we can conclude that *any* valid sequence of n operations (`uba_add` or `uba_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.

8 Exercises

Exercise 1 (sample solution on page 15)**.** *Here's the code for* `uba_new`*:*

```
53 uba* uba_new(int size)
54 //@requires 0 <= size;
55 //@ensures is_uba(\result);
56 //@ensures uba_len(\result) == size;
57 {
58   uba* A = alloc(uba);
59   int limit = size == 0 ? 1 : size*2;
60   A->data = alloc_array(string, limit);
61   A->size = size;
62   A->limit = limit;
63
64   return A;
65 }
```

Line 59 initializes `limit` *to 1 if* `size` *is equal to 0, and to* `size*2` *otherwise — do you see why?*

Let's update our notion of cost to say that, in addition to each array write costing one token (like earlier), allocating an array of n elements costs n tokens. With this new cost model, what is the worst-case cost of `uba_add`*? How many tokens would we need to budget for each* `uba_add` *operation in order to prove that it has a constant amortized cost?*

Exercise 2 (sample solution on page 17)**.** *How would our amortized analysis change if we increased the size of the array by 75% instead of 100%? What if we increased it by 300%? You are allowed to have a cost in fractions of a token.*

Exercise 3 (sample solution on page 17)**.** *From the previous exercise, we see that resizing with larger factors costs fewer tokens than resizing with smaller factors. Why might we choose nonetheless to resize by a smaller factor rather than a larger factor?*

Exercise 4 (sample solution on page 17)**.** *You have recently opened a coffee shop. So far, your coffee shop is still little known, and you have one coffee machine to serve your (few) customers in a timely fashion. You notice that each customer drinks one coffee a day, and they always come back the next day since your coffee is so good. One coffee machine is enough to keep up with 50 people a day. Then, one day, you get 50 people and buy a new machine to serve them. To celebrate, you give a free coffee to all the old customers. And more customers come! You realize that this business model, giving a free coffee to all previous customers each time you expand, will make your coffee shop very popular very quickly!*

The price of a coffee machine is $100 and it costs you $2 to brew each cup of coffee. If you throw a party each time you get a new machine, how much should you charge your customers for this idea to be profitable? Is there a better strategy?

Exercise 5 (sample solution on page 18)**.** *If we only add to an unbounded array, then we'll never have less than half of the array full. If we want* `uba_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* `uba_rem` *operation in order to preserve it? What is the resulting amortized cost (in terms of array writes) of* `uba_rem`*?*

Exercise 6. *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/4` *gives the right amortized cost. At which step in that argument would you notice that* `L->size <= L->limit/2` *is the wrong choice?*

Sample Solutions

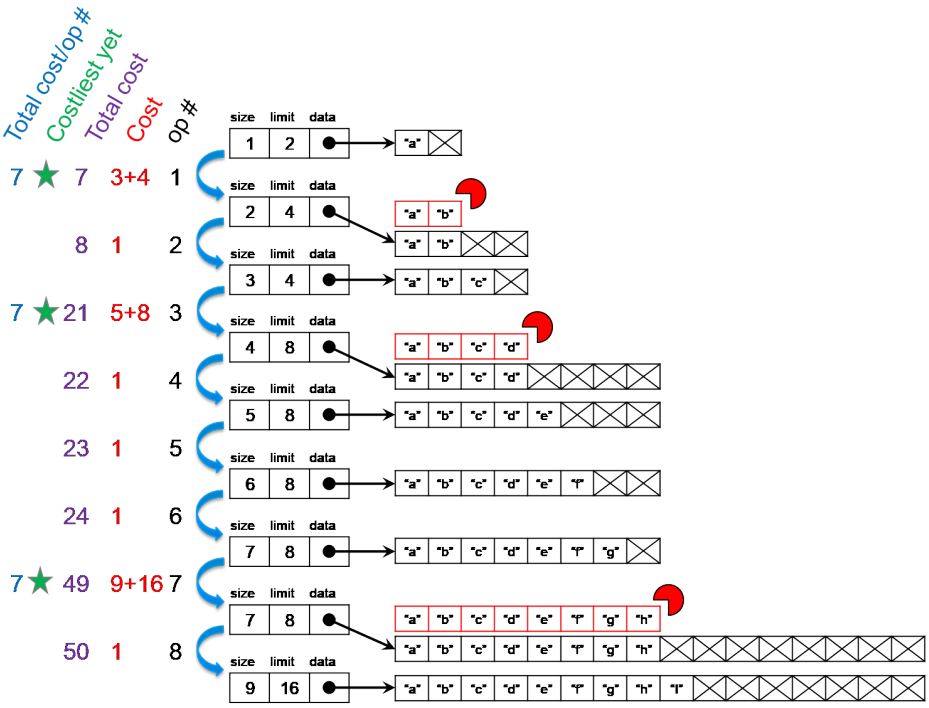
Solution of exercise 1
Were line 59 to be simply `int limit = size*2`, then `limit` would remain at 0 whenever `size == 0`.

Assume the array has limit n and size $n-1$. The next `uba_add` will trigger a resize. We will write the new element (which costs one token), allocate a new array of length $2n$ (which costs $2n$ tokens), and copy the elements of the old array to the new array (which costs n tokens). The total cost is therefore $3n+1$ tokens, which is in $O(n)$.

To compute the amortized complexity of `uba_add`, let's follow our methodology:

- Draw a short sequence of operations.
- Write the cost of each operation.
- Flag the most expensive
- For each operation, compute the total cost up to it.
- Divide the total cost of the most expensive operations by the operation number in the sequence.
- Round up — that's the candidate amortized cost.

The following picture starts with no tokens in reserve, and a length-2 array containing one element. It goes through eight calls to `uba_add`.



For clarity, we split the cost of each operation into the array writing part (which is identical to what we saw earlier) and the allocation part. On this short run, we see that the total cost of the most expensive operations divided by the operation number is 7. This is is the amortized number of tokens we pretend each call to `uba_add` costs.

To show that this number is remains correct for longer runs, assume we have zero tokens in reserve after resizing the array from size k to $2k$. Since each call to `uba_add` writes to the array, we will use one of the 7 tokens obtained from the user right away, leaving a surplus of 6 tokens. Let's see what happens at step $k + i$ for $0 < i \leq k$:

- If $0 < i < k$, we simply save these surplus tokens. Right after step $k + i$, we will have $6i$ tokens in reserve.
- When $i = k$, we need to resize. As seen earlier, we will need to perform one write to the old array, allocate a new array of size $4k$ and copy all $2k$ elements from the old array to the new array. Altogether, we need $6k+1$ tokens. We have $6(k-1)$ tokens in reserve and receive 7 additional tokens from this call to `uba_add`. Now, $6(k-1)+7 = 6k+1$, which is exactly what we need for this step. We are left with zero tokens in reserve as we start the next cycle.

Solution of exercise 2
In both cases, we begin our analysis right after an array resize and assume that we have no reserve tokens leftover.

Resize by 75%: Suppose the array contains k elements on the last resize. After this resize, the length of the array (`limit`) is $k + 75\%k = 7k/4$. The next resize will happen when once we fill this new array, i.e., after $3k/4$ more calls to `uba_add`. To resize it, we need to copy over all $7k/4$ elements into the larger array. So for each of the $3k/4$ added elements, we must receive at least $(7k/4)/(3k/4) = 7/3$ tokens from the caller to have enough tokens to pay for copying all $7k/4$ elements to the new array. We also need one additional token for writing each of the $3k/4$ added elements in the first place into the old array. In total, we need to charge $7/3 + 1 = 10/3$ tokens for each `uba_add` operation.

Resize by 300%: If the array contains k elements on the last resize, it's length is $4k$. So we will resize it again after $3k$ `uba_add` operations. Then, we will copy over all $4k$ elements into the larger array. Thus, for each of the $3k$ elements we add, we need to receive at least $4k/3k = 4/3$ tokens from the caller. We also use another token for writing each of the $3k$ added elements to the old array. In total, we need to charge $4/3 + 1 = 7/3$ tokens for each `uba_add` operation.

Solution of exercise 3
Resizing by a larger factor requires allocating more memory which will remain unused longer. For example, if we resize a length k array by a factor of 2, the new array will contain k empty positions and it will take k calls to `uba_add` to fill the last position. On average, $k/2$ positions will be empty during this time, which amounts to a quarter of the resized array being empty. If we resize by a larger factor, say 4, the new array will contain $3k$ empty positions and it will take $3k$ calls to fill the last position. On average, $3k/2$ positions will be empty during this time, i.e., $3/8$ of the array will be empty on average.

If space usage is the only consideration, smaller resize factors are preferable. However, if space is not too much of a concern for some niche application but we want resizes to be infrequent, then using a larger resize factor may be beneficial.

Solution of exercise 4
Suppose you have n coffee machines and have just reached $50n$ customers: you buy a new machine ($\$100$) and give all $50n$ customers free coffee (this costs you $\$100n$). You will expand again once you get 50 more new customers, which means that these 50 customers must bring in enough income

to pay for the new machine ($\$100$), the expansion party ($\$100n$) and the coffee you are brewing for them $\$2 \times 50 = \$100$). Thus, these next 50 customers must be charged $(\$100 + \$100n + \$100)/50$, i.e., $\$(4 + 2n)$ for each cup of coffee. This depends on n and therefore can get very expensive! Your new business is not likely to go very far if you charge this much for coffee!

You need to throw fewer parties. Inspired by unbounded arrays, let's see what happens if throw a party each time you *double* your number of coffee machines? If you have n coffee machines and $50n$ customers, you purchase n additional coffee machines (at a cost of $\$100n$), give free coffee to your old $50n$ customer (which costs you $\$100n$). But now, it will take $50n$ new customers before you need to buy new machines (and throw a party). These next $50n$ customers will need to bring in enough income to pay for the machines ($\$100n$), the party ($\$100n$) and the coffee they will be drinking themselves ($\$2 \times 50n = \$100n$). To cover these costs, you need to charge each of them $(\$100n + \$100n + \$100n)/50n = \$6$.

You may have noticed that this second scenario matches closely the analysis of `uba_add` for unbounded arrays: each new customer corresponds to adding an element to the array, and each coffee corresponds to a token.

Solution of exercise 5

We assume a cost model where only array writes incur a cost. If our array has length k, `uba_add` resizes it by doubling its length to $2k$ when adding a new element makes it completely full. Instead, `uba_rem` resizes the array to half its size, $k/2$ once it becomes a quarter full, i.e., once it contains $k/4$ elements. In both cases, we need to have enough tokens in reserve to copy all the elements from the old array to the new array: that's k and $k/4$ tokens respectively. Notice that there is no inherent cost to `uba_rem` when we are not resizing the array since nothing gets written in the array in this case.

We can show that `uba_rem` has $O(1)$ amortized cost by charging one token for this operation (each call to `uba_add` continues to cost three tokens) — the amortized cost is constant since the number of tokens charged is constant too.

Let's first look at our worst possible situation: `uba_add` just doubled the length of the array from k to $2k$ and let's assume that copying the elements to the new array consumed all the tokens in reserve. At this point (with k elements and 0 saved tokens), we make a series of $k/2$ calls to `uba_rem`: after the last one, the array will contain $k/2$ elements which is a quarter of its length, $2k$, and therefore it will cause the array to be resized down back to length k. Starting our sequence of `uba_rem` with 0 tokens in reserve, each call adds one token in the savings. So, once the get to the $k/2$-th call, we have $k/2$ tokens in reserve, which is exactly what we need to pay for copying the remaining $k/2$ elements to the new array in the accompanying

resize. We are left with 0 tokens in reserve.

Were we to carry out only `uba_add` or `uba_rem` between resizes, this would be the end of our analysis. But a generic run will intermix them. In this more general setting, we may not consume all saved tokens when resizing the array, but we will always have enough tokens in reserve to pay for copying its elements.

To see this, let's modify the above scenario by inserting one call to `uba_add` followed by one call to `uba_rem` in the sequence of `uba_rem`. By the end of this extended sequence and just before resizing the array, our savings will contain $k/2$ tokens from our original $k/2$ calls to `uba_rem`, but also 2 tokens from calling `uba_add` and 1 token from the additional `uba_rem`. That's 3 tokens more than we need to copy the array elements to resize it.

In general, we may end up with an arbitrary number of surplus tokens in reserve. For example, if right after a resize up we alternate calling `uba_add` and `uba_rem` k times each, we will have accumulated $3k$ tokens in reserve and yet we are far from needing to resize the array.