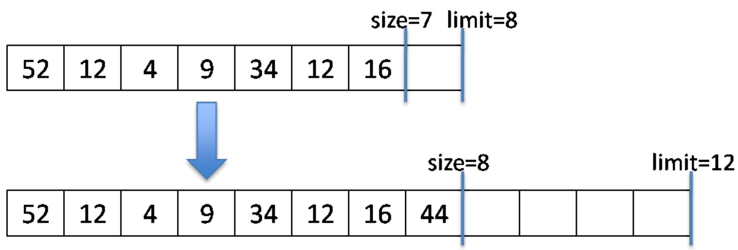
15-122: Principles of Imperative Computation Spring 2023

Recitation 07: Array Disarray **Thursday February 23rd**

Unbounded arrays

When implementing unbounded arrays on an embedded device, a programmer is concerned that doubling the size of the array when we reach its limit may use precious memory resources too aggressively. So she decides to see if she can increase it by a factor of $\frac{3}{2} = 1.5$ instead, rounding down if the result is not an integral number.



This means that it won't make sense for the limit to be less than ________, because otherwise you might resize the array and get an array that wasn't any bigger. This needs to be reflected in the data structure invariant!

```
struct uba_header {
  int size;
  int limit;
  string[] data;
};
typedef struct uba_header uba;

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

bool is_uba(uba* A) {

  ________________________________________

  ________________________________________

  ________________________________________
}
```

Checkpoint 0

Implement the function `uba_resize(uba* A)` for this version of unbounded arrays which resizes the array `A` as described above. Give appropriate preconditions and postconditions, and use an assertion to guard against overflow. (You should not need all the lines provided.)

```
void uba_resize(uba* A)
//@requires ____________________________________________________;
//@requires ____________________________________________________;
//@ensures _____________________________________________________;
{
  if (________________________________) return; // No resizing needed
  assert(__________________________________);   // Failure: can't handle bigger!















}
```

Checkpoint 1

Right after an array resize, we should assume we'll have no tokens in reserve for an array with size k and length $3k/2$ (let's assume k is even).

We might have to resize again after as few as ____________ `uba_add` operations.

That next resize would force us to use ______________ tokens to copy everything into a larger array (with size $9k/4$). The adds that we do in the meantime add elements to the last third of the array.

Each cell in that last third therefore needs to have _________ tokens associated with it.

This gives `uba_add` an amortized cost of _________ tokens, because we need one token to do the initial write whenever we call `uba_add`.

Checkpoint 2

Our analysis indicates that a smaller resizing factor gives us a higher amortized cost, even if it's still in $O(1)$. This indicates that doing n operations on this array, while still in $O(n)$, has a higher constant attached to it. Does this make sense?

____

____

____

You will find in the course of your study in algorithms that, like in this example, achieving higher space efficiency often necessitates a tradeoff in time efficiency, and vice versa.

Checkpoint 3

Repeat this analysis for the case where we triple the size of the array (both if we are only able to use whole tokens and if we're allowed to have an amortized cost of a fraction of a token).

We might have to resize again after as few as __________ `uba_add` operations.

That next resize would force us to use __________ tokens to copy everything into a larger array (with size $9k$). The adds that we do in the meantime add elements to the last $2/3$ of the array.

Each cell in the last $2/3$ therefore needs to have ____________ tokens associated with it.

This gives `uba_add` an amortized cost of ____________ tokens (if fractional tokens are allowed, otherwise _________), because we need one more token to do the initial write whenever we call `uba_add`.