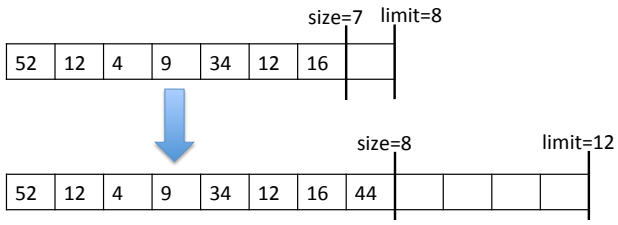
15-122: Principles of Imperative Computation

Recitation Week 6 — Josh Zimmerman

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!

```
1  struct arr_header {
2    int size;
3    int limit;
4    string[] data;
5  };
6
7  bool is_arr_expected_length(string[] A, int limit) {
8    //@assert \length(A) == limit;
9    return true;
10 }
11
12 bool is_uba(struct arr_header* A) {
13
14   ____________________________________________________________
15
16   ____________________________________________________________
17
18   ____________________________________________________________
19 }
```

Checkpoint 0

Implement the `arr_new(n)` function for this version of unbounded arrays, including an `assert()` to guard against overflow like we did in lecture. Make sure that the data structure invariants will be satisfied if `size` is very small. (Can you do a better job of handing very large arrays than we did in lecture?)

Checkpoint 1

Right after an array resize, we should assume we'll have no tokens in reserve and 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 _____ `arr_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 that 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?

Checkpoint 3

Try this analysis if we triple the size of the array (both if we only able to use whole tokens and if we're allowed to have an amortized cost of a fraction of a token). What if we resize by a factor of 5/4 or 4/3? What if we resize by a factor of 13/10?