15-122: Principles of Imperative Computation

Recitation 8 — Josh Zimmerman

Amortized analysis

Amortized analysis lets us consider the runtime behavior of a sequence of operations of an algorithm.

It lets us take a more nuanced view of the runtime of an algorithm: if there's some incredibly rare operation that takes a long time to do, it doesn't make sense to characterize the entire performance of the algorithm by that one operation. By using amortized analysis, we can get a more accurate view of how the algorithm will actually run.

There are a few ways to determine the amortized cost of an algorithm. We'll go over two of them here.

The first, called *aggregate analysis*, is a method in which we first determine that n operations take at *WORST* $T(n)$ steps. Then, we can conclude that the *amortized* cost of the algorithm is

$$\frac{T(n)}{n}$$

There's another method, called the *accounting method*, where we pay some number of tokens for each operation, and can optionally put some tokens aside at each step. Then, we can use the tokens we set aside balance out the cost of a later expensive operation. We look at the total number of tokens we pay over n operations to see what the amortized cost of each operation is.

There's another way of thinking about the accounting method where we start with some number of coins (say, n) and then use that to pay for our operations. We must be able to show that the number of coins we have never goes negative when doing such an analysis. If we show that, then we've shown that the operations we do will have total cost at most n.

You can use either of these methods depending on what you find easiest in a particular situation. Different problems call for different methods of doing amortized analysis.

If these descriptions of methods don't make complete sense to you, don't worry about it yet—we're going to go through a couple of examples during this recitation.

Unbounded arrays

Unbounded arrays are implemented as pointers to `struct uba_headers`:

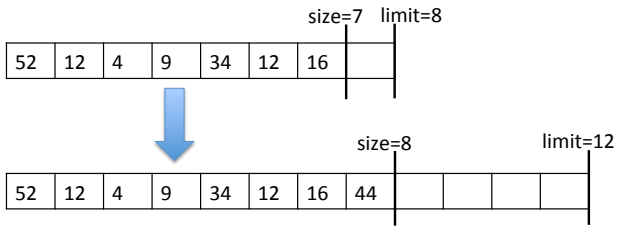
```
1 struct uba_header {
2   int size;
3   int limit;
4   elem[] data;
5 };
```

We're going to discuss a variation on the UBAs presented in class to give a different example of amortized analysis and discuss

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 to have fewer than _____ elements in the array, because otherwise you might resize the array and get an array that wasn't any bigger. This would need to be reflected in the data structure invariant!

We're also going to resize the arrays a little bit earlier than the UBAs we discussed in lecture (which means we may use a bit more memory sometimes!) to make sure that we never end up with a full array.



Make sure you know how to write a data structure invariant for this modified UBA:

```
bool is_uba(struct uba_header* U) {



}
```

We now carry out the the amortized analysis for this version of unbounded arrays. We'll start right after we've resized the array, when the size of the array is s and the limit of the array is $l = 3s/2$. We'll show that, assuming we start out with some non-zero number of tokens k, the next resizing of the array, from size l to size $l' = 3l/2 = 9s/4$, can be paid for by the cost (in tokens) of the operations that happen before that resize. (Just assume s is divisible by 4 for the purposes of this questions.)

Most of the operations require us to spend 1 token because we write to the `U->data` array exactly once.

In addition, we need to reserve _____ tokens for a total amortized cost of _____ tokens.

Starting from a $2/3$ full array, if we write into the old array every time, then after _____ insertions we will fill up the old array completely.

At this point, we have a total of _____ tokens, and we need to copy _____ tokens into the newly allocated array of size $l' = 3l/2$.

After all those copies, we have _____ tokens left. This is no smaller than k, and we have considered the worst case, so we will never run out of tokens.

You can try running through this analysis with other resizing factors, like tripling the size of the array or multiplying it by 5/4, 4/3, or 11/10.

If we were nervous about our analysis, we could add a field `tokens` to our data structure and include the condition that this field must be greater than or equal to 0. That way, if we ever drive our token count below zero, the `is_uba` postcondition will fail.

```
1  void uba_add(uba U, elem e)
2  //@requires is_uba(U);
3  //@ensures is_uba(U);
4  {
5    U->tokens = U->tokens + ____________; /* Amortized cost of uba_add */
6    U->data[U->size] = e; /* Actually write to the array */
7    U->tokens--; /* This write costs us 1 token */
8    U->size++;
9
10   if (U->size == U->limit) {
11     assert(U->size <= (max_int()/3)*2 + 1); /* check for overflow */
12     uba_resize(U, 3*(L->size/2); /* resize, rounding down */
13   }
14 }
15
16 void uba_resize(uba L, int new_limit)
17 //@requires is_uba(L);
18 //@requires L->size < new_limit;
19 //@ensures is_uba(L);
20 //@ensures L->limit == new_limit;
21 {
22   elem[] B = alloc_array(elem, new_limit);
23   for (int i = 0; i < L->size; i++)
24     //@loop_invariant 0 <= i && i <= L->size;
25     {
26       B[i] = L->data[i]; /* Copy over one array element */
27       L->tokens--; /* This copy has cost 1 */
28     }
29   L->limit = new_limit;
30   L->data = B;
31 }
```

Practice!

Unbounded array insertion — aggregate analysis

Using aggregate analysis, show that adding an element to an unbounded array takes amortized $O(1)$ time. We'll only count the number of array writes, for simplicity.

Unbounded array insertion — accounting analysis

Using an accounting analysis, show that adding an element to an unbounded array takes amortized $O(1)$ time (the array starts out empty).

Again, we'll only be counting array writes.

Assume that the limit is n, and that $n > 0$ (the analysis can also work when $n = 0$, but there's an annoying special case).

Binary counter

Consider the situation where we have an n-bit binary number. Assume that flipping a bit (changing it from 1 to 0, or from 0 to 1) is a constant-time operation.

What is the amortized time complexity of incrementing the number, in terms of n?