An Intro to Amortized Analysis

A particularly interesting and useful kind of analysis is called *amortized analysis*. This kind of analysis can be very difficult, but it can be used to solve difficult problems.

Up until this point, we have focussed on the analysis of the cost of specific operations, parameterized by the conditions that the operation will occur in (i.e. analyzing the cost of `FINDKTH` in a list of length $N$).

In amortized analysis, we calculate the total cost of a sequence of operations (or compute the average cost of each operation in a sequence of operations).

- A very simple example: buying in bulk.

  Imagine that you drink iced tea every day at lunch. You can buy individual bottles for $1.50 each, or buy 24 bottles for $20.

  If you buy 24 bottles at lunch one day, this is an expensive operation-- but once you have paid this price, the next 23 days your tea is “free”.

  The cost of buying tea for 24 lunches is $36 if you buy them one-at-a-time, or $20 if you buy them all at once.

  (Of course, there is some overhead involved with storing the tea all that time, and keeping it cold.)
Dynamic Resizing of Arrays

An advantage of linked lists over arrays for implementing list-based data structures is that arrays are of fixed length, while linked lists can grow and shrink dynamically.

Imagine that we decide to use arrays anyway, by doing the following:

- If we try to INSERT, and the array is full, then:
  - allocate a new, larger array - $O(1)$
  - copy the contents of the current array into the new array - $O(N)$
  - deallocate the current array - $O(1)$
  - continue on, using the new array.

- Similarly, if we do a DELETE and find that the array is “mostly” empty, allocate a new, smaller array, etc.

Note: the reallocation of arrays is implemented by the `realloc` function in the standard C library.
Analysis of Dynamic Resizing

Growing/Shrinking by a Constant

If we resize the array by a fixed increment each time (i.e. every time we need a longer array, we extend the length of the array by some constant C), then our algorithm could be quite slow, if the length of the array is large.

Consider the case when the array starts at length 0, and we increment it by 5 every time we run out of space. If we simply add 100 elements to this array, we will have to reallocate the array at 0, 5, 10, 15, 20, ... and each time the array is reallocated, it involves copying 0, 5, 10, 15, 20, ... array elements into the new array.

In general, if we start at 0 elements and then allocate N elements, the allocation of the Nth element costs $O(N)$, because of the last C allocations, C-1 of them cost $O(1)$ (because no extra work was necessary) but one of them cost $O(N)$.

- This is terrible and much too costly if the list grows and shrinks over a wide range. (If it doesn’t grow or shrink very much, if we pick C carefully, this might not be too horrible.)
Growing or Shrinking by a Proportion

Consider the following changes to our algorithm:

- When the array is full, double its size.
- When the array is less than 1/4 full, halve its size.

This changes the analysis of our algorithm considerably.

Consider again the case when we start with an empty array of length 1 and add N elements. We’ll need to reallocate when the array length reaches 2, 4, 8, 16, ...

- The allocation of each element in this case requires $O(1)$ copies \textit{on average}. The allocation of any particular element might be expensive, but this expense is amortized over the cost of the many cheap allocations that must occur between the expensive allocations.
- The more expensive an allocation is, the larger the number of subsequent cheap allocations-- the number of cheap allocations following an expensive allocation is proportional to the cost of that expensive allocation.
Which Proportions to Use?

Note that in this scheme, we shrink the array by half when it is 1/4 full. It would seem more symmetric to shrink the array by half when it is half full-- but that won’t always work. Why not?

- Imagine that the array has just reached size 9, and so has 16 elements. Imagine what happens if the program now enters a loop where it deallocates and allocates an element:
  - after each deallocation, the array has 8 elements and is therefore only half full, so we shrink it by half, requiring 8 copies.
  - now the array is full, so when we attempt to allocate a new element, we need to double its size, requiring another 8 copy operations.

By doubling when the array is full, and halving when the array is 1/4 full, we have a nice property: after either operation, the resulting array is exactly half empty. That means that the number of operations (either element allocation or element deallocation) required to trigger the next reallocation of the array is, at minimum, proportional to the number of elements that were just copied. Therefore, by the time the next reallocation takes place, the cost of the previous reallocation has been amortized for an average cost of $O(1)$. 