1 Amortization

I will start with an amortization analogy: In order to go attend classes at the university you must a sum of tuition. We can think of tuition in two ways. We can think of very first day of class as disproportionally expensive, and the rest of the classes are free to attend; or we can think of this situation as paying tuition up front, and every day of the quarter forward is payed for by a small chunk of tuition. This same idea can be applied to operations on data structures. Consider the implementation of the `add` function on an array list:

```
public class ArrayList {
    int[] data;
    int size;
    ...
    public void add(int element) {
        if (size == data.length) {
            int[] newData = new int[data.length * 2];
            for (int i = 0; i < data.length; i++) {
                newData[i] = data[i];
            }
            data = newData;
        }
        data[size++] = element;
    }
}
```

Given an array list with m elements, where $m < n$, we try to add n elements. At some point we will hit the case where `size == data.length`. In this case, the run time of add is $\mathcal{O}(n)$ (analogous to a new quarter starting). In every other case, add is $\mathcal{O}(1)$. If we were being pedantic, we would have to say that the run time of `add` is $\mathcal{O}(n)$ (which is technically the case in the worst case). This isn't the case in the other $n-1$ `add` operations. What we'd like to do is distribute the work done by the $\mathcal{O}(n)$ operation to the other adds (because they benefit from the resize). In comes amortization:

$$\frac{sum\ of\ runtimes}{\#\ of\ operations}$$

For the example above, we do 1 operation that is $\mathcal{O}(n)$, and $n-1$ operations that are $\mathcal{O}(1)$. To amortize:

$$\frac{\mathcal{O}(n) + (n-1)\mathcal{O}(1)}{(n-1)+1}$$

We can use the definition of big O to drop the o notation and replace the $\mathcal{O}(f(n))$ with $cf(n)$, but I will ignore the constants for simplicity (trust that it will still work):

$$\frac{n+n-1}{n} = \frac{2n-1}{n} \in \mathcal{O}(1)$$

As we can see, the amortized runtime of `add` is $\mathcal{O}(1)$. It's important to include the word "amortized" when we talk about this function, because it while it's annoying to say the worst case is $\mathcal{O}(n)$ it is still true, while saying the worst case is $\mathcal{O}(1)$ is false.